# viinex 3.4

## user's guide

2025

# Contents

**1 Overview** **9**

**2 Configuration** **11**
2.1 Media source objects . . . 14
2.1.1 RTSP video source . . . 14
2.1.2 ONVIF device . . . 17
2.1.3 Media source plugin . . . 19
2.1.4 Raw video source . . . 20
2.1.5 Video channel from a third-party VMS . . . 25
2.2 Media storage and replication . . . 26
2.2.1 Local media storage . . . 26
2.2.2 Recording controller . . . 29
2.2.3 Replication source . . . 30
2.2.4 Replication sink . . . 31
2.2.5 Joint media storage . . . 33
2.2.6 Cloud media storage . . . 34
2.3 Media streaming . . . 37
2.3.1 RTSP server . . . 37
2.3.2 WebRTC server . . . 39
2.4 Media processing and auxiliary objects . . . 42
2.4.1 Video renderer . . . 42
2.4.2 Stream switch . . . 45
2.4.3 Rules . . . 46
2.5 General purpose objects . . . 47
2.5.1 Modbus GPIO device . . . 47
2.5.2 Relational database connection . . . 50
2.5.3 Script . . . 55
2.5.4 External process . . . 57
2.5.5 Web server . . . 61
2.5.6 Publisher for objects in configuration clusters . . . 63
2.5.7 WAMP client . . . 64
2.5.8 Floating license server . . . 66
2.5.9 Metrics . . . 68
2.6 Third-party video management systems . . . 69
2.6.1 Milestone XProtect . . . 69
2.6.2 Geutebrück G-Core . . . 71
2.6.3 Qognify (SeeTec) Cayuga . . . 72
2.6.4 Pelco VideoXpert . . . 73
2.6.5 Bosch BVMS . . . 74
2.6.6 DSSL Trassir . . . 77
2.6.7 Macroscop and Eocortex . . . 78
2.6.8 ITV|AxxonSoft Intellect . . . 79
2.6.9 Native plugins for other VMS integrations . . . 79
2.6.10 Script-driven VMS integrations . . . 81

2.7 Authentication and authorization . . . 82
2.7.1 Role-based access control . . . 82
2.7.2 Token authentication . . . 84
2.7.3 Authentication and authorization provider object . . . 84
2.7.4 Sample configuration of `authnz` object . . . 88
2.7.5 Endpoints implemented by `authnz` object . . . 89
2.8 Common configuration sections . . . 90
2.8.1 RTP transport priority . . . 90
2.8.2 Raw video device operation mode . . . 90
2.8.3 Video encoder . . . 92
2.8.4 Overlay . . . 94
2.8.5 Video renderer layout . . . 96
2.9 Endpoint types . . . 98
2.9.1 `AclClient` . . . 98
2.9.2 `AclProvider` . . . 98
2.9.3 `AclStorage` . . . 98
2.9.4 `ArchiveControl` . . . 99
2.9.5 `ArchiveVideoSource` . . . 99
2.9.6 `ArchiveVideoSourceClient` . . . 99
2.9.7 `AuthnzClient` . . . 99
2.9.8 `AuthnzProvider` . . . 99
2.9.9 `ClusterManagerReader` . . . 99
2.9.10 `ClusterManagerProvider` . . . 99
2.9.11 `ConfigReader` . . . 99
2.9.12 `DynamicVideoClient` . . . 100
2.9.13 `DynamicVideoSource` . . . 100
2.9.14 `EventArchive` . . . 100
2.9.15 `EventSink` . . . 100
2.9.16 `EventSource` . . . 100
2.9.17 `KeyValueStore` . . . 100
2.9.18 `KeyValueStoreClient` . . . 100
2.9.19 `LayoutControl` . . . 101
2.9.20 `MetaConfigStorage` . . . 101
2.9.21 `MetricsConsumer` . . . 101
2.9.22 `MetricsExporter` . . . 101
2.9.23 `MetricsProducer` . . . 101
2.9.24 `OverlayControl` . . . 101
2.9.25 `PtzControl` . . . 101
2.9.26 `RawVideoSource` . . . 102
2.9.27 `RawVideoSourceClient` . . . 102
2.9.28 `RecControl` . . . 102
2.9.29 `RecControlClient` . . . 102
2.9.30 `ReplicationSink` . . . 102
2.9.31 `ReplicationSource` . . . 102
2.9.32 `ServicePublisher` . . . 102
2.9.33 `SnapshotSource` . . . 103
2.9.34 `Stateful` . . . 103
2.9.35 `StreamSwitchControl` . . . 103
2.9.36 `TimelineProvider` . . . 103
2.9.37 `Updateable` . . . 103
2.9.38 `VideoSource` . . . 103
2.9.39 `VideoSourceClient` . . . 103

2.9.40 `VideoStorage` . . . 103
2.9.41 `VideoStorageClient` . . . 104
2.9.42 `VmsConnection` . . . 104
2.9.43 `VmsConnectionClient` . . . 104
2.9.44 `WebRTC` . . . 104
2.10 Links . . . 104
2.10.1 Syntax . . . 104
2.10.2 Semantics . . . 106
2.10.3 Link types . . . 107
2.11 Variables interpolation . . . 107
2.12 License information . . . 108
2.12.1 Local license document . . . 109
2.12.2 Floating license client . . . 109
2.12.3 Provisional licenses . . . 110
2.13 Split configuration . . . 113

**3 HTTP API 117**
3.1 Web server . . . 117
3.1.1 Enumerate published components . . . 117
3.1.2 Obtain the metainformation on published components . . . 118
3.2 Authentication . . . 119
3.2.1 Authentication challenge . . . 119
3.2.2 Authentication response . . . 120
3.3 Environment . . . 122
3.3.1 Attached SenseLock USB dongles . . . 122
3.3.2 License document content . . . 122
3.3.3 Probe for licenses . . . 124
3.3.4 Obtain viinex 3.4 software version . . . 126
3.3.5 Discover visible ONVIF devices . . . 126
3.3.6 Probe an ONVIF device . . . 128
3.3.7 Discover connected raw video sources . . . 131
3.4 Video source . . . 134
3.4.1 Status information . . . 134
3.4.2 Live stream . . . 134
3.5 Video archive . . . 135
3.5.1 Status and statistics . . . 136
3.5.2 Archive contents . . . 137
3.5.3 Disk usage for a specific time interval . . . 138
3.5.4 Overall disk usage for a specific time interval . . . 139
3.5.5 Media export . . . 139
3.5.6 Media playback . . . 141
3.5.7 Remove records from video archive . . . 142
3.6 Recording controller . . . 143
3.6.1 Status information . . . 143
3.6.2 Change recording status . . . 144
3.6.3 Flush accumulated video data to disk . . . 145
3.7 Managed replication . . . 146
3.7.1 Enqueue a new replication task . . . 146
3.7.2 Get information on replication task . . . 148
3.7.3 Manage status of replication task . . . 150
3.7.4 Remove a replication task . . . 151
3.7.5 Enumerate all replication tasks . . . 152

3.7.6 Get the timeline from a VMS channel . . . 152
3.8 Snapshots . . . 153
3.8.1 Get a snapshot from the snapshot source . . . 153
3.9 Overlay . . . 156
3.9.1 Clear overlay . . . 156
3.9.2 Change overlay bitmap . . . 156
3.9.3 Change overlay HTML . . . 157
3.10 Video renderer . . . 158
3.11 Layout control . . . 158
3.11.1 Get the names of linked video sources . . . 158
3.11.2 Set the layout for the video renderer . . . 158
3.11.3 Set the background color or background image . . . 160
3.11.4 Set or clear the image for viewports of disconnected video sources . . . 161
3.12 Stream switch . . . 163
3.12.1 Get the names of linked video sources . . . 163
3.12.2 Switch to a specific stream . . . 163
3.13 PTZ control . . . 164
3.13.1 Get the PTZ node description . . . 164
3.13.2 Get presets . . . 167
3.13.3 Create a preset . . . 167
3.13.4 Remove a preset . . . 168
3.13.5 Update a preset . . . 169
3.13.6 Go to a specified preset . . . 170
3.13.7 Update the “home” position . . . 170
3.13.8 Go to the “home” position . . . 171
3.13.9 Get the coordinates of a current position . . . 172
3.13.10 Move the PTZ device . . . 172
3.13.11 Stop the PTZ motion . . . 174
3.14 WebRTC signaling . . . 174
3.14.1 Obtain a general information on WebRTC server . . . 174
3.14.2 Create a new session . . . 175
3.14.3 Media data request format . . . 177
3.14.4 Provide an SDP answer for a session . . . 178
3.14.5 Update an existing session . . . 180
3.14.6 Get session status . . . 181
3.14.7 Gracefully shutdown a WebRTC session . . . 182
3.15 Event storage . . . 183
3.15.1 Get the summary for events stored in the database . . . 183
3.15.2 Retreive events from the database . . . 184
3.16 Authentication details and ACL-related storage . . . 186
3.16.1 Enumerate users accounts . . . 187
3.16.2 Get the ACL version . . . 188
3.16.3 Get the users-to-roles mapping . . . 188
3.16.4 Get the access control entries . . . 189
3.16.5 Get all ACL information in one call . . . 190
3.16.6 Modify users accounts . . . 191
3.16.7 Modify an access control list . . . 193
3.17 Abstract interfaces . . . 195
3.17.1 Stateful . . . 195
3.17.2 Updateable . . . 196
3.18 WebSocket interface . . . 197
3.19 Configuration clusters . . . 199

3.19.1 Enumerate existing clusters . . . 200
3.19.2 Create a new cluster of objects . . . 200
3.19.3 Remove an existing cluster of objects . . . 201
3.19.4 Enumerate components published by a cluster . . . 202
3.19.5 Obtain the metainformation on components published by a cluster . . . 202
3.19.6 Access viinex 3.4 objects in configuration clusters . . . 203
3.19.7 Obtaining events from a cluster . . . 204
3.20 Scripting-style API endpoint . . . 204

**4 Scripting and JS API** **208**
4.1 Execution model and handlers . . . 208
4.1.1 `onload` . . . 209
4.1.2 `ontimeout` . . . 209
4.1.3 `onevent` . . . 209
4.1.4 `onupdate` . . . 210
4.1.5 Example . . . 210
4.1.6 Asynchronous operations and anonymous callbacks . . . 212
4.2 General purpose functions . . . 213
4.2.1 `vnx.publish()` . . . 213
4.2.2 `vnx.timeout()` . . . 214
4.2.3 `vnx.timer.delay()` . . . 214
4.2.4 `vnx.event()` . . . 215
4.2.5 Logging . . . 216
4.2.6 `require()` and modules . . . 216
4.2.7 Linked objects . . . 217
4.2.8 Configuration clusters . . . 218
4.2.9 Local filesystem . . . 219
4.2.10 HTTP client . . . 220
4.2.11 Auxiliary functions . . . 223
4.3 Application interfaces . . . 224
4.3.1 `LayoutControl` . . . 224
4.3.2 `PtzControl` . . . 225
4.3.3 `RecControl` . . . 225
4.3.4 `SnapshotSource` . . . 226
4.3.5 `Stateful` . . . 227
4.3.6 `StreamSwitchControl` . . . 228
4.3.7 `TimelineProvider` . . . 228
4.3.8 `Updateable` . . . 228
4.3.9 `VideoStorage` . . . 229
4.3.10 `WebRTC` . . . 230
4.4 Script-driven VMS integration API . . . 230
4.4.1 Live media stream request . . . 231
4.4.2 Archive media stream request . . . 232
4.4.3 Get credentials for HTTP client . . . 233
4.4.4 Get the snapshot image . . . 233
4.4.5 Obtaining the timeline . . . 235

**5 WAMP interface** **238**
5.1 Naming convention . . . 238
5.2 Calling convention . . . 239
5.3 Remote calls . . . 240
5.3.1 Service interfaces . . . 241

5.3.2 Application objects . . . 242
5.3.3 Managing configuration clusters . . . 243
5.4 Events . . . 244

**6 Native API** **246**
6.1 Brief C and C++ API overview . . . 246
6.2 Acquiring raw video by means of local transport . . . 248
6.3 Implementing the H264 video source plugin . . . 249
6.4 Implementing the VMS integration plugin . . . 250

**7 Deployment** **252**
7.1 Installation . . . 252
7.1.1 Windows . . . 252
7.1.2 Linux . . . 254
7.1.3 Running viinex 3.4 in foreground . . . 255
7.1.4 Setting the number of OS threads . . . 255
7.2 License key management . . . 256
7.2.1 Obtaining information on attached USB dongles . . . 256
7.2.2 Obtaining the license document from a USB dongle . . . 257
7.2.3 Obtaining information on PC hardware . . . 257
7.2.4 Updating a license document on the USB dongle . . . 258
7.2.5 Working with an “emulated” license storage . . . 258
7.2.6 Batch mode . . . 259

**References** **260**

This document is a reference manual to help software developers in building applications with viinex 3.4.

If you have technical questions on viinex 3.4, please contact the support team at Viinex helpdesk: https://viinex.atlassian.net/servicedesk/customer/portal/1/

Compiled on June 16, 2025.

# 1 Overview

viinex 3.4 is a software development kit (SDK) for adding video surveillance and video management features into applications which would benefit from having them. viinex 3.4 implements functionality for receiving media data from external devices (IP cameras and encoders, USB cameras), storing media in the archive, re-streaming video to clients. It has means to receive and store events for cameras builtin video analytics, run custom video analytics on the server.

viinex 3.4 implements WebRTC for media streaming from live media sources and media archives. This makes viinex 3.4 suitable for applications which require low latency live streaming. Most importantly, WebRTC provides the direct media transport between viinex 3.4 and client software, even if each of these parties reside in different private networks behind a NAT.

In its interaction with applications, particularly in video data interchange, viinex 3.4 sticks to ISO-standardized media formats. When dealing with H.264 video codec, viinex 3.4 provides access to recorded video in such formats as MP4 [1], MPEG TS [2], and in form of raw H.264 stream which is also handy for a number of video processing applications. When it comes to streaming video data to the client, viinex 3.4 implements HLS specification [3]. Internally, viinex 3.4 stores and manages video archive as a sequence of MP4 files named and arranged across subfolders, according to the video origin and timestamps, in a transparent and obvious manner. This allows a user, in case of necessity, to operate on a video archive by standard means, such as Windows built-in media player (for example if media containing video archive is detached from the device where viinex 3.4 was installed and brought to another standard PC with no additional software).

There is a media replication functionality available in viinex 3.4, which can be used to either automatically replicate data between viinex 3.4 media archives, or to pull the data from an arbitrary source into viinex 3.4 media archive upon request.

viinex 3.4 implements an integration with a AWS S3 (and compatible services) to provide a media storage for media archive. This makes it possible to have a media archive of an arbitrary depth. Also S3-backed media archive allows to share recorded media data between many viinex 3.4 instances.

There is also an extensible support for third-party video management systems in viinex 3.4. VMS integrations can be used to re-stream video from a 3rd party system in HTML5-compatible manner, to explore the contents of an external video archive, to request for snapshots from the integrated VMS, and to replicate a video archive data from that VMS to viinex 3.4 video archive for further streaming and processing.

For implementing custom logic like events processing, control video recording, video analytics and so on, viinex 3.4 has a builtin scripting capabilities. JavaScript language may be used to extend the logic implemented by viinex 3.4.

There are programming interfaces available for simple, robust and efficient integration of other 3rd party video analytics modules with viinex 3.4.

viinex 3.4 is designed to be embeddable, and does not bring its own end-user interface to the product where it is used. viinex 3.4 is completely separated from customer's application

address space; there's no need for linking your code with viinex 3.4 client libraries. All interaction with viinex 3.4 is conducted via HTTP or WAMP programming interfaces, either of which can be reached from wide range of programming languages.

# 2 Configuration

viinex 3.4 instance can be run with one or more configuration files of simple JSON format, which sections' semantics is described below. The configuration for viinex 3.4 is a JSON document (or several documents, see section 2.13) containing three optional keys: `objects`, `links` and `license`.

The first two define the functionality of the system being instantiated. From overall view, objects within viinex 3.4 are the elementary building blocks of the system: the object is created and being run as a whole. On the other hand, each object may represent different subset of functionality. For instance, an "RTSP video source" may represent only a live video source and a source of live snapshots, while the "Video channel from a 3-rd party VMS" represents a video source, a source of live snapshots, but also a source of video archive data (archive stream), and a source of video archive timeline information. Another example: an object of the same type, "ONVIF device", depending on object's configuration may represent a source of live video, a source of events, and a PTZ control, – and these three subsets of functionality do not depend on each other.

For that purpose, the concept of an *endpoint* is distinguished within Viinex architecture. An endpoint may be viewed as a substitute for an interface from object-oriented programming paradigm, or, somewhat broader, a contract. An *endpoint type* means a type of such contract, that is conventions on which functionality the component provides, and/or which behavior it demonstrates. Each object exposes one or more *endpoints* of different types. An endpoint may be represented as a set of methods being implemented within viinex 3.4 and exposed via API, but not necessarily: a component which does not expose methods but uses other components' endpoints in a certain way, i.e. behaves as a client to these endpoints, can be classified as such that exposes the (client) endpoint. Respective examples are the "Recording controller" component, which makes it possible for other components, including external software, to control the process of video recording to an archive, – this functionality is exposed as endpoint of type `RecControl`, – and the "Rule" object which implements the logic for turning video recording on and off, based on events being caugt, – and for this purpose it uses an endpoint of type `RecControl`, it implements the logic to use a recording controller, and thus it says to implement the endpoint of type `RecControlClient`. Note that while `RecControl` endpoint definitely exposes certain methods via the API, – this is required to actually turn the video recording on and off, – the `RecControlClient` has no methods to expose, it cannot be used directly via API, but the object which implements an endpoint of respective type, demonstrates the behavior of a recording control client.

Configuration document syntax should be as follows:

```
{
    "objects":
    [
        {
            "type": "TYPE1",
            "name": "NAME1",
            "meta": JSON_VALUE_1,
            "parameter1": "value1",
```

```
            ...
            "parameterN1": "valueN1"

        },
        …
        {
            "type": "TYPE_M",
            "name": "NAME_M",
            "meta": JSON_VALUE_M,
            "parameter1": "value1",
            ...
            "parameterNM": "valueNM"

        }
    ],
    "links":
    [
        ["NAME11", "NAME12"],
        ...
        ["NAME_k1", "NAME_k2"]
    ],
    "license": "LICENSE_DOCUMENT_STRING"
}
```

The `objects` section is a JSON array for elementary functionality units ("objects", or module instances) to be run when viinex 3.4 is started. Each unit's configuration is a JSON object having two mandatory fields, `type` and `name`, one optional field `meta`, and a number of other fields that can be mandatory or optional, depending on functional unit's type.

In its turn, functional unit's type, which is defined by the `type` parameter in unit's configuration object, determines the set of programming interfaces implemented by the unit, or, in other words, the type of functionality which is exposed by the unit to other units created in viinex 3.4 (via internal interfaces), and to the user (via HTTP API).

The `name` field of unit's configuration object acts as a label for referring to this unit in `links` section. The value of `name` property also affects the runtime behavior of the system, in particular it influences the URLs for addressing this unit via HTTP API, names of folders for storing runtime data, etc.

The optional `meta` field may hold an arbitrary JSON value, including a JSON object. The purpose of that field is to tag the viinex 3.4 component with some information coming from the application which uses viinex 3.4. The latter reads the value from `meta` property of configuration objects, and may report it via corresponding request in HTTP API, see section 3.1.2.

Now, while the `objects` section of configuration defines which objects (components) should be created, and specifies the details for their behavior, – the `links` of configuration section defines which objects should interact with each other. Within this model, objects interact according to the aspects of their behavior, and every such aspect is formalized as an `endpoint type`. The `links` section defines objects, pairwise, which should interact with each other. The semantics of such interaction is inferred from which endpoints are exposed by each of the object in each pair. A link can only be established between objects which have at least one matching

endpoint[1].

The first three sections of this chapter describe the configuration of particular objects. For each object type, a reference information is given on endpoint types that can be exposed by such object. Section 2.9 briefly describes the contract assumed by endpoints of each particular type. Section 2.10 describes which types of endpoints can be linked together, and how respective link is implemented.

An example for this file is shown below:

```
{
    "objects":
    [
        {
            "type": "rtsp",
            "name": "cam1",
            "meta": { "desc": "Backyard" },
            "url": "rtsp://192.168.0.121:554/ISAPI/streaming/channels/101",
            "auth": ["admin","12345"],
            "transport": ["mcast","tcp"]
        },
        {
            "type": "rtsp",
            "name": "cam2",
            "meta": { "desc": "Hall" },
            "url": "rtsp://192.168.0.111:554/ISAPI/streaming/channels/101"
        },
        {
            "type": "storage",
            "name": "stor0",
            "meta": { "desc": "Long-term storage", "volume": "/dev/sdb" },
            "folder": "/home/viinex/videostorage",
            "filesize": 16,
            "limits": {
                "max_size_gb": 10
            }
        },
        {
            "type": "webserver",
            "name": "web0",
            "port": 8880
        }
    ],
    "links":
    [
        [["cam1", "cam2"], "stor0"],
        ["web0", ["cam1", "cam2", "stor0"]]
    ],
```

[1]There's currently a limitation in viinex 3.4 configuration syntax and semantics: when two objects are linked, – viinex 3.4 attempts to match all endpoints implemented by a first object with all endpoints implemented by a second object, and all matched endpoints are linked. There's no way to prevent endpoints of a specific type from linking, despite semantics of object linking suggests that links are established between endpoints.

```
    "license": "rqZ821uWtcsxz....fYZE76ByAgmCZO"
}
```

## 2.1 Media source objects

### 2.1.1 RTSP video source

viinex 3.4 implements the RTSP [4] client for accessing H.264 live video streams [5, 6] sent by IP video cameras or third party RTSP servers. Configuration object for RTSP video source in viinex 3.4 is denoted by unit type `rtsp`. Such configuration object should contain one mandatory field, `url`, and optional fields `auth`, `transport`, `dynamic` and `rtpstats`. An example for RTSP video source configuration is given below:

```
{
  "type": "rtsp",
  "name": "cam1",
  "url": "rtsp://192.168.0.121:554/ISAPI/streaming/channels/101",
  "auth": ["admin","12345"],
  "transport": ["tcp", "udp"],
  "interleaved_any": false,
  "rtpstats": true,
  "dynamic": true
}
```

or, another example, suitable with some IP cameras,

```
{
  "type": "rtsp",
  "name": "cam1",
  "host": "192.168.0.121",
  "port": 554,
  "auth": ["admin","12345"]
}
```

The `url` field is a string containing RTSP URL to connect to. The URL can optionally contain port information (in form of `address:port`).

An alternative to specifying a RTSP URL is setting the parameter `host` and, optionally, the parameter `port`. Setting the `host` parameter alone is quivalent to setting the RTSP URL of value `rtsp://HOST:554/`. Setting both the `host` and `port` parameters is equivalent to setting the RTSP URL of value `rtsp://HOST:PORT/`. Such settings can be convenient with certain equipment like IP cameras. In general case, however, setting just the `host` and `port` is not sufficient to access an RTSP source, so the usage of the property `url` is recommended.

The `auth` property, if present, should be a pair of login and password, — the credentials to be used for accessing the RTSP server. If `auth` element is not specified, viinex 3.4 tries to access specified RTSP URL without authentication.

The `transport` property, if present, should be a list of transport-layer protocols to be used by RTP protocol when obtaining the video data. This list specifies client's preferences with

respect to RTP transport. Format for this member is described in paragraph 2.8.1. performed. If `transport` parameter is not specified, viinex 3.4 RTSP client defaults to `"transport": ["udp", "tcp"]` which effectively enables UDP unicast and TCP (in that order of preference) but disables UDP multicast.

Somewhat connected with `transport`, the optional `interleaved_any` property instructs viinex 3.4 to ignore possibly incorrect interleaved channel IDs sent by some RTSP servers. When it set to `true`, viinex 3.4 would accept all RTP data sent over RTSP connection in interleaved mode, not filtering packets according to `interleaved` header attribute previously negotiated with server. A default value for this option is `false`.

The optional `dynamic` property, when set to `true`, instructs viinex 3.4 that the video stream from an RTSP source should only be acquired while video is requested by clients via

- an RTSP server (when an external client makes a connection to the RTSP server and requests for video stream from this video source),
- a recording controller[2] (while the video recording of this RTSP video source is requested),
- a Web server[3] (for HLS streaming of video from this source, when an external HTTP client requests for the HLS stream of this video source),
- a WebRTC server (when an external client makes a connection to WebRTC server and requests for video stream form this video source),
- a video renderer (while this video source is rendered on the current layout).

If the `dynamic` property is set to `true`, viinex 3.4 dynamically establishes the connection to the RTSP video source when the one of the above activities starts, and disconnects from the origin when all of the above activities end, until one of them is started again. In other words, `dynamic` set to `true` saves the bandwidth, only requesting the video data from the origin when it's needed.

Otherwise, if `dynamic` is set to `false`, viinex 3.4 continously acquires the video stream from the specified video source, even if no other object in configuration currently requires that stream. This is also the default behaviour, if the `dynamic` property is not set.

Another difference that makes the `dynamic` parameter is that with that set to `true`, the respective RTSP video source does not require a dedicated viinex 3.4 license for connecting the videochannel. Instead, the license lease is acquired dynamically when the video stream is requested by one (or several) clients, and released when that stream is no longer needed. This makes it possible to add more video sources to the configuration then it is specified in the license document, as long as it is known that no more than the licensed quantity of video sources are ever requested simultaneously.

The `rtpstats` property, when set to the value `true`, instructs the RTSP video source to gather statistics on received and lost RTP packets and payload units (which are NAL units in case of H264 video payload). Default value for this property is `false`. The statistics gathered in this

[2]Note that prerecording feature of the recording controller won't work as expected with the dynamic video sources, so it's recommended that prerecording is set to 0 for such use cases.

[3]Because of the nature of HLS protocol, specifically the need for having some pre-buffered data prior to HLS streaming can be started, it should be kept in mind that an HLS client will experience a significant delay before it receives the first MPEG TS fragment of HLS stream from viinex 3.4 dynamic video source. Some HLS client implementations are not robust enough and give up prematurely, before said delay elapses and the needed video data gets pre-buffered. If this is the case, an additinal effort might need to be taken at client's side to take this delay into account.

way is periodically reported by `rtsp` object in the form of events of special topic `RtpStats` (for more information on receiving events generated by viinex 3.4 objects please refer to section 3.18). More specifically, events generated by an RTSP video source object to report RTP statistics have the form of

```
{
  "topic": "RtpStats",
  "timestamp": TIMESTAMP,
  "origin": {
    "type": "rtsp",
    "name": STRING
  },
  "data":{
    "since": TIMESTAMP,
    "till":  TIMESTAMP,
    "packets": {
      "received": INTEGER,
      "lost": INTEGER
    },
    "frames": {
      "received": INTEGER,
      "lost": INTEGER
    }
  }
}
```

Here, the `topic`, `timestamp` and `origin` are the properties common for all events. The `origin.name` contains the name of an object (RTSP video source) where the RTP statistics data originates from. The properties `data.since` and `data.till` specify the time interval when the statistics was gathered. The structures `packets` and `frames` contain the numbers of received and lost RTP packets and NAL units, respectively.

Note that while the `received` figures represent the accurate number of datagrams received and of NAL units completely reassembled by RTP parser, the `lost` values, when not equal to 0, should be treated as a lower estimate of the actual number of lost packets/frames[4]

No matter whether the `rtpstats` property is set to `true` or not, an RTSP video source uses the mechanism of events to report on RTSP connection errors. In particular, the event of the following form is generated if a RTSP connection failure occurs:

```
{
  "topic": "RtspException",
  "timestamp": TIMESTAMP,
  "origin": {
    "type": "rtsp",
    "name": STRING
```

[4]This is because viinex 3.4 accounts for packets as `lost` if it clearly expects those packets and they are necessary to reassemble a frame from an RTP stream, and such packets do not arrive. If a frame consists of one RTP packet which is lost (or of several packets which are all lost), – viinex 3.4 cannot infer how many frames were lost and does not account for such packets and frames. Moreover, if viinex 3.4 RTP parser sees that a frame cannot be reassembled because of some portion of packets in frame was lost, – it does not attempt to analyze which exact number of packets is lost for that frame, – but only accounts for the first portion of lost RTP datagrams. This, however, gives a statistically appropriate estimate with network losses rate up to 5%.

```
  },
  "data":{
    "exception": STRING
  }
}
```

The `topic` property is equal to the value `RtspException` for such events. Their `data.exception` holds the textual description of an error occured.

Endpoint(s) implemented by objects of type `rtsp`:

- 2.9.38 `VideoSource`,
- 2.9.33 `SnapshotSource` (unless the `dynamic` property is set),
- 2.9.16 `EventSource` (when the `rtpstats` property is set),
- 2.9.13 `DynamicVideoSource` (when the `dynamic` property is set),
- 2.9.23 `MetricsProducer`.

### 2.1.2 ONVIF device

viinex 3.4 supports ONVIF specification for retrieval of H.264 and H.265 video streams, G.711 (u-law and a-law), G.726, AAC and Opus audio streams, and events from ONVIF devices. Configuration object for ONVIF device in viinex 3.4 is denoted by unit type `onvif`. Such configuration should contain exactly one of two mutually exclusive mandatory fields: `url` or `host`. The `url` parameter, if given, should contain the URL of `onvif/device_serice` SOAP service, as reported in `xaddrs` array in the result of ONVIF discovery call described in section 3.3.5. As an alternative, the `host` parameter may be given instead of `url`. The `host` property should contain an IP address of ONVIF device or a DNS name resolvable to such IP address. Along with the `host` parameter, the optional `port` parameter may be given to specify the TCP port which ONVIF Device service is listening on at the target device. If the `port` property is not given, the default value 80 is assumed.

There is an optional parameter `enable`[5] which is a JSON array and may hold from zero to three elemetns — `"video"`, `"audio"`, `"events"`, and/or `"ptz"`. The purpose of this parameter is to instruct the instance of current object to acquire or not acquire the data of respective type from the ONVIF device, or to expose or not to expose the PTZ functionality via the API. By default, if the `enable` parameter is omitted, it is assumed that both video and events should be acquired (but PTZ is not used by default, and neither is audio streaming enabled by default). For instance, if `"enable":["video"]` is specified, only the video data will be obtained. This is sometimes important to disable the attempts to subscribe for ONVIF events, if some specific camera does not implement this functionality of misbehaves upon receiving subscription requests. Note that to use PTZ functionality, one has to explicitly enable it, probably along with video, audio and events, like this:

```
...
"enable": ["video","audio", "events","ptz"],
...
```

[5] Formerly this parameter had the name `acquire`. The name `acquire` is still valid for backward compatibility, but is deprecated.

An optional `rtpstats` parameter may be specified to instruct an ONVIF device object to gather network statistics while receiving video via RTSP/RTP protocol. The RTP statistics is gathered when the property `rtpstats` is set to the value `true` in configuration, and is reported by the `onvif` object as the stream of events which can be obtained from viinex 3.4 via WebSocket interface described in section 3.18. The syntax and semantics of respective events is discussed in section 2.1.1. The default value for `rtpstats` property is `false`.

Like with the RTSP video source (see section 2.1.1), the RTSP-related errors are reported using the events mechanism, irrelevant to the value of property `rtpstats`.

There can also be the `dynamic` parameter specified for the configuration of the ONVIF device object. The meaning of this parameter exactly matches that for the RTSP video source, as described in section 2.1.1. Note that `dynamic` parameter for ONVIF device object only affects the video streaming. Events and PTZ control, if enabled in configuration, are always enabled.

Another three optional parameters which can be specified with both `url` and `host` variants are `auth`, `transport` and `profile`. The `auth` element, if present, should be a pair of login and password, — the credentials to be used for accessing ONVIF API via SOAP and later the RTSP video endpoint. The `transport` property, if present, should contain a list of preferred RTP transport protocols for receiving video data. The syntax for `auth` and `transport` parameters is the same as for that parameters in RTSP video source configuration described in section 2.1.1 and 2.8.1.

Last but not least, there is an optional `profile` parameter which specifies the "token" (unique identifier within the device) of the profile to be used. If this parameter is not set, viinex 3.4 automatically selects the first[6] available profile on the device that is set up for H.264 streaming.

Two examples for "ONVIF device" object's configuration is given below, for URL variant:

```
{
  "type": "onvif",
  "name": "cam1",
  "url": "http://192.168.0.111/onvif/device_service",
  "auth": ["admin", "12345"],
  "acquire": ["video", "audio", "events"],
  "profile": "Profile_2",
  "transport": ["tcp"],
  "dynamic": false
}
```

and for the host/port variant, with the very minimum set of parameters required:

```
{
  "type": "onvif",
  "name": "cam1",
  "host": "192.168.0.111"
}
```

"ONVIF device" module implements two logical interfaces within Viinex: the interface of live video source, and the interface of event source. The latter can be used for instance to control the video recording process via rules, see section 2.4.3.

[6]The order is actually defined by the device. viinex 3.4 takes the first appropriate profile that is described in the result to `GetProfiles` ONVIF call [9].

The implementation of event source interface of "ONVIF device" module in viinex 3.4 does not require any settings. viinex 3.4 automatically creates pull-point subscription and begins to acquire all events produced by ONVIF device, – right after an instance of the module is started.

Endpoint(s) implemented by objects of type `onvif`:

- 2.9.38 `VideoSource` (when video is enabled),
- 2.9.33 `SnapshotSource` (when video is enabled),
- 2.9.16 `EventSource` (when events are enabled),
- 2.9.25 `PtzControl` (when PTZ is enabled),
- 2.9.13 `DynamicVideoSource` (when the `dynamic` property is set),
- 2.9.23 `MetricsProducer`.

### 2.1.3 Media source plugin

In order to provide applicaitons with capability to make arbitrary video streams available as live video sources in viinex 3.4, the latter supports the functionality of so-called `mediasourceplugin`.

The idea behind that functionality is that an application can provide a shared library that implements certain simple API, which is described in section 6.3, calls the specified factory method in that library, which should return a specific implementation of a video source[7]. That media source is then used by viinex 3.4 just like any other video source.

The configuration for the live media source plugin should look as follows:

```
{
    "type": "mediasourceplugin",
    "name": STRING,
    "dynamic": BOOLEAN,
    "library": STRING,
    "factory": STRING,
    "init": JSON
}
```

where property `type` should take the value of ``mediasourceplugin''; value of property `name` identifies the video source in viinex 3.4 configuration; and the property `dynamic` means exactly the same as it does for RTSP video source or an ONVIF device: it means that a video stream would only be consumed by viinex 3.4 while that stream is requested by someone.

The properties `library`, `factory` and `init` are specific to the plugin though. The `library` property specifies which shared library (a .dll "dynamically linked library" on Windows or a .so "shared object" on Linux) should be loaded by viinex 3.4 in order to instantiate the plugin. In its turn, the `factory` property is the symbol inside of the loaded dynamic library which is looked for by viinex 3.4 and, once found, is called, in order to instantiate the pluggable video source.

[7] To be precise, the `vnxvideo_media_source_t` type and the C++ interface `IMediaSource` represent a media source co-located with event source, so that a plugin can produce events synchronized with an audio/video stream. For more details refer to section 6.3.

The `init` property serves as the means to pass an initialization information to the plugin. It could be an arbitrary JSON value, but most convenient use of it is an object. That value is serialized into string by viinex 3.4 prior to the calling of `factory` function inside of the `library`. The stringified `init` value is actually passed by viinex 3.4 as an argument to the `factory` call. The plugin implementation can parse that string to JSON value and interpret this value to initialize the behavior of the newly instantiated pluggable video source.

The API for implementing such plugins is published by Viinex Inc. under MIT license and is available at https://github.com/viinex/vnxvideo. That repository also contains an example implementation of the video source plugin, namely – a plugin the read video track from a media file using the `avformat` library (a part of FFmpeg project). Respective source code resides at https://github.com/viinex/vnxvideo/blob/master/src/FileVideoSource.cpp. The `vnxvideo` library is a part of viinex 3.4, and therefore it is present with every viinex 3.4 installation. The `FileVideoSource.cpp` is compiled into the `vnxvideo` library, which makes the file video source plugin available. The configuration of such plugin would look like:

```
{
     "type": "mediasourceplugin",
     "name" : "cam2",
     "library" : "vnxvideo.dll",
     "factory" : "create_file_media_source",
     "init" : {
         "file": "D:\\temp\\videofile.mp4"
     }
}
```

In this example, the plugin implementation is loaded from `vnxvideo.dll`, and the name of plugin factory function is `create_file_media_source` (compare to https://github.com/viinex/vnxvideo/blob/ab0009b/src/FileVideoSource.cpp#L542). The `init` parameter in this case is a JSON object containing one property `file`, for the path to media file, which is deserialized and used in the plugin implementation, as can be seen in the source file referenced above.

Endpoint(s) implemented by objects of type `mediasourceplugin`:

- 2.9.38 `VideoSource`,
- 2.9.33 `SnapshotSource` (unless the `dynamic` property is set),
- 2.9.16 `EventSource`,
- 2.9.13 `DynamicVideoSource` (when the `dynamic` property is set).

### 2.1.4 Raw video source

viinex 3.4 is capable of dealing with raw video sources via programming interface provided by the operating system. The object type for raw video source in viinex 3.4 is `rawvideo`. An example for configuration section for this kind of objects is given below:

```
{
    "type": "rawvideo",
    "name": "raw0",
    "capture": {
```

```
        "type": "dshow",
        "address": "\\\\?\\usb#vid_045e&pid_0779&mi_00#7&b53bc93&0&0000\
                        #{65e8773d-8f56-11d0-a3b9-00a0c9223196}\\global",
        "mode": {
            "pin":"Capture",
            "colorspace":"YUY2",
            "framerate":10.0,
            "limit_framerate": true,
            "size":[1280,720],
            "exposure":"auto"
        }
    },
    "analytics": ["basic"],
    "overlay": [
        {
            "left": 50,
            "top": 100,
            "colorkey": [255, 255, 255],
            "initial": "C:/temp/overlay.html"
        },
        {
            "left": 350,
            "top": 200,
            "colorkey": [255, 255, 255]
        }
    ],
    "encoder": {
        "type": "cpu",
        "quality": "small_size",
        "profile": "high",
        "preset": "ultrafast"
    }
}
```

There are two mandatory subsections of configuraiton object for raw video source – `capture` and `encoder`, – and two optional sections, `analytics` and `overlay`.

### Capture parameters

The `capture` subsection describes the origin where the raw video data should be taken from:

```
    "capture": {
        "type": "dshow" | "v4l" | "localtransport",
        "address": STRING |
        "index": INT,
        "mode": OBJECT
    }
```

The `type` property of `capture` subsection is a string which can take one of three values: `"dshow"` for DirectShow devices on Windows, `"v4l"` for Video for Linux 2 devices on Linux,

or "localtransport" for raw video retreival from other processes using viinex 3.4 local transport mechanism.

The address property should be a string value of address (absolute path) of the device to connect to. On Windows, for "dshow" capture type, the format of address depends on device driver, and it typically similar to one of the following:

```
\\?\usb#vid_045e&pid_0779&mi_00#7&b53bc93&0&0000
                        #{65e8773d-8f56-11d0-a3b9-00a0c9223196}\global,

\\?\pci#ven_1131&dev_7133&subsys_52010000&rev_f0#5&2b491bae&0&0000f0#
{65e8773d-8f56-11d0-a3b9-00a0c9223196}\{bbefb6c7-2fc4-4139-bb8b-a58bba724083}
```

but may completely differ with some vendors' drivers. Note that backslashes '\' within the path should be escaped (i.e. each backslash should be doubled) when written to JSON document. On Linux, for "v4l" capture type, video device address typically looks like

```
/dev/video0,
/dev/video1,
```

and so on. The address of a device can be found out by means of raw video device discovery call to viinex 3.4 API documented in section 3.3.7.

Alternatively, for dshow or v4l capture types, instead of address, an index may be specified. index is an integer interpreted as an ordinal zero-based number of the device that viinex 3.4 should connect to, an ordinal with respect to the list of available raw video devices, as returned in response to discovery call described in section 3.3.7. The option to specify a device index instead of an address is useful for creating a pre-defined configuration which does not depend on absolute path to device, because on Windows the path contains vendor and model (and sometimes device serial number) information and therefore cannot be just copied across servers without additional adjustment. Specifying an index instead of address allows for simple copying of configuration files. On the other hand, the order in which devices are listed by the operating system, is not defined, therefore specifying an address of the device is the only way to ensure that the device of specific vendor and model, plugged into specific USB or PCI slot, will get the specific identifier (name) in viinex 3.4.

The mode property of capture structure should have the form described in section 2.8.2. mode property is optional. If not specified, viinex 3.4 automatically uses the first element of capabilities array reported for corresponding device by raw video device discovery call, as described in section 3.3.7. However it is recommended that mode is explicitly specified in the configuration, because this is the only way to obtain video with predictable resolution. There are also some concerns about colorspace to be selected. In order to perform video encoding, viinex 3.4 needs to have video in YUV colorspace in planar format with 4:2:0 subsampling [14]. This is connected with video encoder implementation. That is, an "internal" raw video format for viinex 3.4 is "I420". viinex 3.4 performs the conversion to required raw video format automatically, if mismatching format is chosen in video source configuration. However such conversion takes additional CPU resources, especially if what is converted is not only pixel layout in memory or subsampling ("format"), but also the colorspace itself (i.e. if original colorspace is RGB). It is strongly recommended that selected mode instructs the device driver to operate with video in YUV colorspace, if such capability is available.

A special case of the capture type is "localtransport". This type serves for the purpose of getting raw video from an external process, when the frames are published by means of

viinex 3.4 local transport mechanism. That is, some other software component on the same host may produce raw video frames and use the local transport server component from the `vnxvideo` library to publish these video frames. viinex 3.4 instance, in its turn, may use a local transport client component to connect to a respective local transport server, get raw video frames, and further treat them as if they came from a video camera.

When the `capture.type` property is set to `"localtransport"`, the `capture.address` property should be specified. The `address` should contain an address of the local transport "rendezvous point" – that's the string value passed to `vnxvideo_local_client_create` and `vnxvideo_local_server_create` functions. The parameters `index` and `mode` are ignored for the capture of type `localtransport`.

### Encoder

Video encoder is always created and associated with raw video source by viinex 3.4: raw video is never transmitted over network or written to video storage; it is always encoded first. The optional `encoder` subsection of the raw video source configuration is described in paragraph 2.8.3.

### Analytics

The `analytics` section defines the set of video detectors (analytics) applied to the video acquired from the souce being configured. The `analytics` value should be an array of JSON objects, each defining the settings for an instance of video detector, or an array of strings which is a simplified way of creating the detector of specific type with default settings.

Recognized analytics type is `basic`; it instructs viinex 3.4 to instantiate the algorithms for detection of motion on the video, as well as situations of image quality degradation. The settings section for basic analytics have the following syntax:

```
...
  "analytics": [
    {
      "type": "basic",
      "roi": [LEFT, TOP, RIGHT, BOTTOM],
      "framerate": INTEGER,
      "too_bright": BOOLEAN,
      "too_dark": BOOLEAN,
      "too_blurry": BOOLEAN,
      "motion": FLOAT,
      "scene_change": BOOLEAN
    },
    ...
  ]
```

All parameters except the `type` are optional. The `roi` specifies the rectangular region of interest for the algorithms (in relative units in range $[0.0, 1.0]$. The default `roi` is the whole image. The `framerate` parameter determines the frame rate at which a data acquired from the source should be processed by the analytics. For `basic` detectors, the default and recommended value for framerate is 5. Valid range for this parameter is $[2, 30]$. Note that one should not set the `framerate` value of the analytics module higher than the effective framerate of the source

(after it is limited), because otherwise motion detector may work worse than expected, – but it's safe to set it to lower values to save CPU resources.

The boolean values `too_bright`, `too_dark`, `too_blurry`, `scene_change`, define whether corresponding detectors should be activated within this instance of analytics module. If the value `too_dark` is set to false, the detector of insufficient lighting conditions is set to be inactive, and corresponding events are never produced by this analytics instance. The default value for all of the mentioned parameters is `true`. For motion detector, the floating-point `motion` parameter defines the sensitivity for that detector, with the default value of 0.5, and valid range $[0.0, 1.0]$. The value `"motion": 0.0` effectively turns the motion detector off.

There can be several instances of the analytics module defined for one `rawvideo` object. For that, the `analytics` array of the configuration should have more than one element in it. This can be useful defining a motion detector on several ROIs with different sensitivity in each.

There is also a simplified syntax for creating the analytics module, – simply put the string type of the analytics module instead of the JSON object with its settings:

```
...
  "analytics": ["basic"],
...
```

is equivalent to creating the analytics module of type `basic` with all parameters set to their default values.

The effect of creation of an analytics module instance with the raw video source is that such video source exposes an interface of an event source and can be used in rules (to turn the video recording on and off). Its events can also be acquired via WebSocket interface of viinex 3.4. The syntax and semantics of events generated by `rawvideo` object with video detectors turned on matches that of events acquired from ONVIF devices, see sections 2.4.3, 3.18 for more details.

### Overlay

The `overlay` subsection is optional. If present, it enables the overlay functionality for the raw video source in viinex 3.4. The syntax and meaning of the properties under the `overlay` configuration section are given in paragraph 2.8.4 of this document.

Endpoint(s) implemented by objects of type `rawvideo`:

- 2.9.38 `VideoSource`,
- 2.9.26 `RawVideoSource`,
- 2.9.33 `SnapshotSource`,
- 2.9.16 `EventSource`,
- 2.9.24 `OverlayControl`,
- 2.9.13 `DynamicVideoSource` (when the `dynamic` property is set).

### 2.1.5 Video channel from a third-party VMS

For the purpose of obtaining of live video streams and replication of video recordings into its own video archive, viinex 3.4 supports integration with a number of video management systems from third-party vendors.

In viinex 3.4 such integration introduces two types of objects, – the video management system itself, and a VMS channel. The first object should be viewed as a logical connection to a third-party video management system. It can be a connection to a single server, but it sometimes can be also a connection to the "federation" of VMS servers of the same vendor. Either case, the VMS object in viinex 3.4 is a convenient way to store access credentials that are to be used by all VMS channels. An internal implementation also uses this VMS object to share some resources like HTTP/HTTPS connections pool, etc.

Each VMS object is represented in viinex 3.4 configuration with a JSON object of the form specific to that particular video management system. The configuration format for such objects is described in section 2.6.

The second part of third-party VMS integration to viinex 3.4 is a VMS channel. This is another object type, not specific to any particular VMS, but common for all of them. The configuration of this kind of objects should look as follows:

```
{
    "type": "vmschan",
    "name": "chan1",
    "channel": {"id":"rEMSOtFL"},
    "enable": ["video"],
    "dynamic": true
}
```

The `type` of this object should be equal to `vmschan`. The common `name` parameter should be present. There could be also parameters `dynamic` and `enable` which exactly match the meaning of such parameters as described in section 2.1.2.

The most important and most specific parameter for the `vmschan` object is `channel`. This property represents the channel selector for this VMS channel within the VMS. There are several possible forms for the value of `channel` parameter:

```
"channel": {"id": STRING} |
"channel": {"name": STRING} |
"channel": {"global_number": NUMBER} |
"channel": STRING |
"channel": NUMBER
```

The meaning of each predicate type depends on the VMS, but typically the `"id"` and `"name"` selector predicates are available and mean the match for logical VMS channel id and VMS channel human readable name, respectively. The last two forms of syntax for `channel` property, when the property has the value of a string or of an integer number, are equivalent to specifying `"channel": {"id": STRING}` and `"channel": {"global_number": NUMBER}`.

Particular third-party VMS integrations may support, besides specifying `channel.id` or `channel.name` or `channel.global_number`, also the stream selector and certain additional properties for channel identification. When this is the case, such additional properties need to be

specified in the JSON object syntax of `channel` field value. The stream selector can be specified by setting an additional property `stream`, identifying a substream from the video source. There can also be an optional value `hint` holding an arbitrary JSON value to be parsed and interpreted by the VMS driver. The most generic syntax for channel selector look as follows:

```
"channel": {
  "id":STRING |
  "name": STRING |
  "global_number": NUMBER,
  "stream": null | "main" | "sub",
  "hint": JSON
}
```

with `stream` and `hint` parameters being optional.

It depends on a particular third-party VMS and its integration into viinex 3.4 whether such additional properties are supported. Please refer to section 2.6 for more information.

Note that the `vmschan` object is not specific to any brand of VMS; it is rather an abstract representation for a VMS "channel" (usually mapped to a video camera connected to the third-party VMS). In order to give the flavour and implementation to the objects of type `vmschan`, each of them should be linked with one object which represents a VMS connection (for example, an object of type `milestone`, see section 2.6):

```
  "links": [ ...
    ["milestone1", ["chan1", "chan2", ...]],
  ... ]
```

The absence of such connection for any of `vmschan` objects would result in an error during viinex 3.4 startup.

The objects of type `vmschan` can also be used in the links with other viinex 3.4 objects, in all contexts where alse the objects of type `onvif` could be used. A `vmschan` may also be used as a source for managed video replication; see section 3.7 for more details.

Endpoint(s) implemented by objects of type `vmschan`:

- 2.9.38 `VideoSource`,
- 2.9.5 `ArchiveVideoSource`
- 2.9.36 `TimelineProvider`,
- 2.9.33 `SnapshotSource`,
- 2.9.43 `VmsConnectionClient`.

## 2.2 Media storage and replication

### 2.2.1 Local media storage

viinex 3.4 implements the functionality for writing, storing and accessing media data received from external sources.

Configuration object for media archive in viinex 3.4 is denoted by object type `storage.local`[8]. Such configuration object should contain three mandatory fields: `folder`, `filesize` and `limits`. An example for video archive configuration is given below:

```
{
  "type": "storage.local",
  "name": "stor0",
  "folder": "/var/spool/viinex/videostorage",
  "filesize": 16,
  "limits": {
    "max_size_gb": 500,
    "max_depth_abs_hours": 480,
    "max_depth_rel_hours": 240,
    "keep_free_percents": 5
  },
  "allow_removal": false
}
```

The `folder` element specifies the locally mounted filesystem and path where video data is stored and maintained.

> **NB!** Local video storage implementation assumes that it has exclusive access over the folder provided in its configuration. While an instance of local video storage is running, no external modifications to the filesystem under that folder are allowed. This includes the prohibition for manual operations over files within that folder, as well as for running another instance of local video storage configured to use the same folder. Breaking this assumption results in undefined behavior.

The element `filesize` specifies an approximate limit for stored video fragments' size, in Megabytes. This parameter affects performance and memory usage in various scenarios; for more details contact Viinex technical support.

The element `limits` is JSON object containing four optional fields:

- `max_size_gb`,
- `max_depth_abs_hours`,
- `max_depth_rel_hours` and
- `keep_free_percents`.

The parameters listed above affect the behavior of video archive with respect to maintaining the "ring" of video records. Video archive automatically removes older video files as new ones are recorded, in order to enforce the limits on disk space which is used by video data. There are four parameters allowed under `limits` section of video archive configuration to specify various strategies for that purpose. All of mentioned parameters are optional; if none of them is given or `limits` section is absent, then no limits on stored video data size are enforced. When multiple parameters under `limits` section are present, they all act simultaneously (which

[8] Prior to viinex version 3.4 this object type was known as simply `storage`. That name is retained for backwards compatibility.

leads to enforcing the most strict limit, as they are evaluated in runtime, by the moment of evaluation). The semantics of parameters under `limits` section is as follows:

`max_size_gb` parameter sets the limit for total disk space utilized by video storage. To enforce this limit, the video storage iteratively removes the oldest video record, until total disk space used by video data becomes less than the value specified by this parameter. This is the most widely used parameter under `limits` section.

`max_depth_abs_hours` parameter sets the temporal limit with respect to system clock. All video records older than the value of this parameter are removed. The maximum video record's age is set as a number of hours (integer). Note that this limit acts even if no new data is written to video storage. It is useful mainly to enforce some limits that might be imposed by legislator requirements (for instance those connected with privacy policies).

`max_depth_rel_hours` parameter, in contrast to the previous parameter, sets the temporal limit (in hours) with respect to the most recent video data which appears in the video archive. This option helps keeping required "recording depth", and is especially useful in scenarios where video data replication is performed, so measuring "recording depth" from system clock is inappropriate. Note that the time of the "most recent" data is computed over all video sources maintained by the instance of video storage, so if at some point a video data from many sources is stored in the video archive, but the new data originating from only *one* source is appended to the archive, — still, oldest data to be removed would be selected over *all* sources in order to enforce this limit. (This is a common rule anyway: one instance of video storage always removes video data from all sources; removed video is the one which is older than some single (that is — same for all stored video sources) point in time).

`keep_free_percents` parameter helps preserving the viinex 3.4 from completely using up all available disk space, which may render the system unusable in some situations. The value for this parameter is specified in percents. If it is given, the video archive instance removes oldest video records, until at least specified fraction of disk space is freed. The actual space to be freed by viinex 3.4 on a volume is computed taking into account the disk space already used by other applications. That is, if total volume size is $T$, and disk space used by other applications is $S$, — then the disk space ever available to viinex 3.4 is $T - S$, and the actual space specified in `keep_free_percents` is computed in runtime from $T - S$ rather than $T$. The actual figure is re-estimated periodically, which allows viinex 3.4 video storage to play nicely with other applications that can be actively using and freeing space on the same volume.

The `allow_removal` boolean property indicates whether this instance of storage would allow external applications to arbitrarily remove data upon API request. Respective API call is described in section 3.5.7. If not specified, this option is set to `false` which means that recordings are only removed from storage automatically in order to enforce specified `limits`; specifically, – only the oldest data is always removed, while the most recent data is preserved. The `DELETE` API call 3.5.7 is disabled for such instances of video storage. If the `allow_removal` property is set to `true`, the `DELETE` API call is enabled.

Endpoint(s) implemented by objects of type `storage.local`:

- 2.9.39 `VideoSourceClient`,
- 2.9.4 `ArchiveControl` (when the `allow_removal` property is set), and
- 2.9.40 `VideoStorage`, which is actually a factory for the following endpoints not published implicitly but can be looked up and accessed by `ServicePublishers`:
- 2.9.5 `ArchiveVideoSource`,

- 2.9.33 `SnapshotSource`,
- 2.9.36 `TimelineProvider`,
- 2.9.23 `MetricsProducer`.

### 2.2.2 Recording controller

viinex 3.4 implements the functionality to control the process of recording a video data to the video archive. For that, the recording controller object is employed.

Recording controller represents a logical "switch" that has two positions: "recording started" and "recording stopped". The controller, despite its name, does not itself take a decision to change the position of that switch, but rather manages data streams, based on position of the switch. Decisions to start and stop the recording are taken by either rules, or an external software (via HTTP remote procedure calls to the recording controller, in the latter case). For more information see sections 2.4.3, 3.6.

An instance of recording contoller is denoted in viinex 3.4 configuration by object type `recctl`. Video sources and the video archive associated with the recording controller are specified in `links` section of the configuration. An example of configuration object for recording controller is given below:

```
{
  "type": "recctl",
  "name": "rec0",
  "prerecord": 5,
  "postrecord": 3
}
```

Two parameters need to be set for the recording controller. The `prerecord` field defines the minimum length of so called pre-recording buffer, in seconds. Pre-recording buffer is a buffer in RAM, and its content is permanently (as new video data is received) renewed to hold at least specified number of seconds of most recent video. (The actual buffer length may be greater than specified because of the need for holding all preceding frames of a first GOP intersecting with specified time interval). When the command to start the recording arrives, the recording controller first sends a content of the pre-recording buffer to the storage, and then continues to pass to the storage the live video data as it appears. This effectively allows to record a fragment of video preceding the moment when the command for the recording has arrived.

> **NB!** The `prerecord` parameter is ignored for the dynamic video sources, i.e. for the RTSP video sources and ONVIF devices with the parameter `dynamic` set to `true`.

Similarly, the `postrecord` field extends the recording for a specified number of seconds past the receiving of a command to stop writing. In contrast with pre-recording, post-recording feature does not require a buffer in RAM.

Note that `prerecord` and `postrecord` parameters are applied to all video sources associated with an instance of recording controller. Same is true for the video recording logic itself: one instance of recording controller has one logical "switch" to turn the recording on and off, and that "switch" affects the recording of all video sources associated with that controller. This

allows for grouping of video cameras into *scenes*, if there is a demand to start and stop the recording simultaneously over all cameras within a scene.

Endpoint(s) implemented by objects of type `recctl`:

- 2.9.39 `VideoSourceClient`,
- 2.9.41 `VideoStorageClient`,
- 2.9.12 `DynamicVideoClient`,
- 2.9.28 `RecControl`.

### 2.2.3 Replication source

viinex 3.4 implements replication functionality split into two parts: replication source and replication sink. Replication source is responsible for sending video data from viinex 3.4 instance where that data is produced/collected, to some other instance where replication sink should be deployed. Replication source connects to its peer replication sink, negotiates with it on what video data is required to be transmitted, and sends this data. Then these steps are repeated. When it turns out that no new data is to be transmitted from replication source to replication sink, the source pauses its activity for some amount of time. The replication source plays active role in data replication.

Replication source is specified in configuration document by an object of type `replsrc`. An example for configuration of the configuration of replication source is given below:

```
{
  "type": "replsrc",
  "name": "replsrc0",
  "sink": "http://10.4.7.12:8881/v1/svc/replsink1",
  "key": "agentA",
  "secret": "foobarsecret42"
}
```

The only parameters to configure in this section define the URL and credentials for accessing the replication sink which the video data should be transferred to. Corresponding credentials (in the above example API key “agentA” with secret “foobarsecret42”) should be known by web server where replication sink is exposed, and corresponding API key should be known by that replication sink; see section 2.2.6 for more details.

Other required parameter for replication source is the video archive instance, to which this replication source is attached and where the video data is taken from. Corresponding video archive is set in `links` section of configuration document. If video archive is not set for replication source, viinex 3.4 gives an error and refuses to start.

Note that replication source treats equally all video channels present in video archive this replication source is attached to. It tries to send video data from all video channels accessible within connected video archive to its peer replication sink. If some of video channels need not to be replicated, they should be written to other video archive, which is not connected with a replication source.

Endpoint(s) implemented by objects of type `replsrc`:

- 2.9.31 `ReplicationSource`,
- 2.9.41 `VideoStorageClient`.

### 2.2.4 Replication sink

Replication sink is viinex 3.4 component that is responsible for accepting the video data from replication sources and storing it into video archive. One replication sink is capable of dealing with multiple replication sources, making it possible to gather video from multiple video archive into one archive on a separate host.

Replication source is denoted in configuration document by an object of type `replsink`. An example of replication sink configuration is given below:

```
{
  "type": "replsink",
  "name": "replsink1",
  "mode": "rolling" | "managed",
  "workers": 8,
  "translation": [
    ["agentA", "site1."],
    ["agentB", "site2."]
  ]
}
```

Two parameters need to be set up for replication sink: `mode` and `translation`. The `mode` parameter should take one of two values, `rolling` or `managed`. The fundamental difference between these two modes is that rolling replication mode is an automatic replication that is performed on behalf of a replication source at some remote instance of viinex 3.4, and all data that is available to the replication source is copied to the replication sink. The managed replication mode, in contrast, means that the whole replication process is controlled via API, see 3.7. The source for replication could be an RTSP source or a channel in some 3rd party VMS (see section 2.1.5).

The configuration of replication sink object is below discussed for these two modes.

#### Rolling replication

The rolling mode means that replication sources attached to this replication sink will send the new video data as soon as it appears in their video archives, not awaiting for explicit orders or requests from replication sink. The `translation` parameter is an array of string pairs, where first element of each pair references an API key of HTTP client, where the second key sets the prefix for camera names. In the whole, it works as follows. When a replication source connects to a replication sink, it always uses some authentication credentials, that is an API key and a secret. That API key is, among other purposes, used by replication sink to lookup the `translation` map and determine an object prefix — a string which is prefixed to camera name reported by replication source, to be stored in video archive local to replication sink.

Given the above example, if a replication source connects to the replication sink using API key `agentA`, and sends the video data from its local cameras with names `cam1` and `cam2`, these video sources will receive the names `site1.cam1` and `site1.cam2` in the video archive that

is attached to replication sink. If, for instance, another replication source uploads the data from `cam1` and `cam2` too, but acts on behalf `agentB`, — that video source will receive names `site2.cam1` and `site2.cam2` at replication sink's side. This mechanism makes it possible to keep object names at replication sources' side simple and typical yet not unique across all replication sources (such as `cam1, cam2`), while preventing the data from different cameras from mixing at replication sink side.

It's up to user who performs viinex 3.4 deployment to choose camera names and prefixes for replication. If camera names are chosen to be unique across all replication sources, the prefixes may be left blank in `translation` section of replication sink configuration. This would provide uniform naming for video sources in all video archives (original and replicated).

At the same time, even if prefixes are left blank, it is required that all the API keys that are used by replication sources to communicate with replication sink, are listed in `translation` section of replication sink configuration. The presence of corresponding API keys in `auth` section of `webserver` configuration (see section 2.5.5 is necessary but not sufficient for the replication sink to accept the data from a replication source.

The `workers` property is ignored by a replication sink in rolling mode.

To accomplish replication sink configuration, two links should be established in `links` section of configuration document: one link with a video archive, and another link with a web server where the replication sink should be exposed. For security reasons, one may decide to create a dedicated web server for that purpose, assigning a unique TCP port for that, especially if the replication sink is exposed on the Internet to accept video data from remote viinex 3.4 instances. Doing so is a normal use case; corresponding configuration example is provided in stock configuration files coming with viinex 3.4 distribution for Windows.

#### Managed replication

Managed replication sink is actually an agent which accepts replication tasks via API descibed in section 3.7, and executes those tasks. Each replication task for managed replication sink represents the instructions on where to take video data from, which time interval that data should be assigned to, which channel in video archive the data should be placed into, and so on.

It is important that replication tasks take their time and computational resources and network bandwidth to execute. Therefore they are, in general case, not executed all at once, in parallel. Managed replication sink introduces the notion of workers, which can be seen as independent threads of execution of replication tasks. Each replication task is executed by a dedicated worker. While a worker executes some replication task, it is busy and cannot execute any other task. After a replication task is completed (or failed), the worker which executed that task returns to the pool of free workers. The new replication tasks are placed in a queue; in their turn, the free workers take the tasks from that queue to execute them.

The property `workers` in replication sink configuration specifies the number of workers for this replication sink instance. This value should be agreed with the content of viinex 3.4 license document which is provided by Viinex licensor. There is a separate license position in the license document which specifies the maximum allowed number of replication sink workers. The actual number of workers is specified by the value `workers` in viinex 3.4 configuration, but it should not exceed the number allowed in the license document, otherwise viinex 3.4 won't be able to start such configuration cluster.

Since the destination for video data replication (the channel within the video archive) is gov-

erned by replication tasks, the `translation` property is ignored by replication sink in managed mode.

Just like for the rolling mode, a replication sink should be linked (in the `links` section of the configuration) against video storage object, and the web server object. The link with a video storage specifies where the incoming video data will be stored. The link with a web server allows the latter to publish the replication sink instance to make its API available for calling by external software components.

The API of replication sink in managed mode is in details described in section 3.7 of this document.

Endpoint(s) implemented by objects of type `replsink`:

- 2.9.30 `ReplicationSink`,
- 2.9.6 `ArchiveVideoSourceClient`,
- 2.9.41 `VideoStorageClient`.

### 2.2.5 Joint media storage

Joint media storage is a proxy object which provides a convenience interface to the users of the system in cases when there are several underlying storages, and the user is interested in combining media data from them. That is, if there is `stor1` which contains media data written on Monday, and the `stor2` which contains media data written on Tuesday, – a user might be interested in seeing the combination of these two, without knowing where exactly each particular fragment of media data resides.

For such cases, the joint media storage object is implemented. Its configuration is simple:

```
{
    "type": "storage.joint",
    "name": STRING,
    "priority_storage": STRING
}
```

As the matter of fact, most important part of configuration in this case is in the `links` section. Joint storage object needs to be linked with "real" storages. Then it knows data from which media storages needs to be combined, when requested by a user. The setting `priority_storage` defines the identifier of one linked storage which is treated as preferrable, when media data for a requested interval is present in more than one underlying storages.

One application of a joint media storage can be to combine media archives with different depths (and, typically, different quality), so that they are presented to the user as a single storage.

Another possible application of joint storage object is combining the local and cloud storages (see next section).

Endpoint(s) implemented by objects of type `storage.joint`:

- 2.9.41 `VideoStorageClient`,
- 2.9.40 `VideoStorage`, which is actually a factory for the following endpoints not published implicitly but can be looked up and accessed by `ServicePublishers`:

- 2.9.5 ArchiveVideoSource,
- 2.9.33 SnapshotSource,
- 2.9.36 TimelineProvider.

### 2.2.6 Cloud media storage

Cloud media storage is an object to store and provide access to media data. It uses a service compatible with AWS S3 for storing media data. It does not use files on a locally mounted filesystem.

The configuration for cloud media storage object looks as follows:

```
{
    "type": "storage.cloud",
    "name": STRING,
    "bucket": STRING,
    "region": STRING,
    "path": STRING,
    "access_key": STRING,
    "secret_key": STRING,
    "filesize": NUMBER,
    "allow_removal": BOOLEAN
},
```

The meaning of these parameters is straightforward. `bucket` should contain the name of S3 bucket which is going to be used for storing data. `region` defines the AWS region for accessing the bucket. Value `path` should contain the prefix which is going to be prepended before the names of media objects. It should end with a slash. Values `access_key` and `secret_key` are optional and define the credentials for accessing S3 account. If these values are not provided, viinex 3.4 uses the user's environment to infer them. Parameter `filesize` should be set to 16 (but actually is ignored). Flag `allow_removal` determines whether this instance of storage is going to provide an API for removal of data from the S3 bucket.

Cloud media storage object is in certain aspects similar to local video storage, however there are substantial differences.

- Provided that S3 buckets are global entities, – cloud video storage does not imply exclusiveness on the underlying bucket. It is designed to share the same bucket with many instances of cloud media storage. This contrasts with implementation of local media storage, which assumes exclusive control over its folder on locally mounted filesystem.
- Cloud media storage object does not have the `limits` section in its config, and does not attempt to enforce any limits on media data stored in the S3 bucket against which it is configured. It is user's responsibility to define the policies for the limits in cloud storage and implement a service to enforce them.
- Cloud media storage object can be used for media playback, however it cannot be used as a destination for video recording in real time. Instead, it can be used as a destination for media replication in "rolling" mode (see section ).

The latter means that in order to have recorded media in the cloud storage, – a user needs to set up a local storage first, and set up the recording into that storage. It is possible to allocate insignificant amount of disk volume for such storage. Minimum requirements may depend on the speed of connection to S3 service. If this connection is not a bottleneck, then local storage should be set up to hold at least one hour of video for each attached media source. Also, the cloud storage should be set up. Then, the rolling replication needs to be set up between the local media storage and cloud media storage. A sample configuration to implement this is provided below:

```
{
    "objects":
    [
        {
            "type": "onvif",
            "name": "cam1",
            ...
        },
        {
            "type": "onvif",
            "name": "cam2",
            ...
        },
        {
            "type": "onvif",
            "name": "cam3",
            ...
        },
        {
            "type": "onvif",
            "name": "cam4",
            ...
        },
        {
            "type": "storage.local",
            "name": "stor0",
            "folder": "/viinexvideo",
            "filesize": 16,
            "limits": {
                "keep_free_percents": 20
            }
        },
        {
            "type": "storage.cloud",
            "name": "storS3",
            "bucket": "viinex-cloud-storage-0",
            "region": "eu-central-1",
            "access_key": "AKIA----REDACTED----",
            "secret_key": "..redacted..",
            "path": "demo/",
            "filesize": 16
        },
        {
```

```
            "type": "storage.joint",
            "name": "storJoint",
            "priority_storage": "stor0",
            "meta": {"name": "Joint storage"}
        },
        {
            "type": "replsrc",
            "name": "replsrc.stor0"
        },
        {
            "type": "replsink.rolling",
            "name": "replsink.storS3"
        },
        {
            "type": "webserver",
            "name": "web0",
            ...
        },
        {
            "type": "webrtc",
            "name": "webrtc0",
            ...
        }
    ],
    "links":
    [
        ["web0", "webrtc0"],

        [["cam1","cam2","cam3","cam4"],"stor0"],

        ["stor0", "replsrc.stor0"],
        ["storS3", "replsink.storS3"],
        ["replsrc.stor0","replsink.storS3"],

        ["storJoint", ["stor0", "storS3"]],

        [["cam1","cam2","cam3","cam4","storJoint"],["web0","webrtc0"]]
    ]
}
```

Here, permanent recording is set up from four cameras into the local archive. A rolling replication is set up from the local archive into S3-backed media storage. Furthermore, joint media archive is set up, linked with both local and cloud media storage, with priority given to local storage. Web server and WebRTC servers are given the access to media sources and only the joint media storage.

Endpoint(s) implemented by objects of type `storage.cloud`:

- 2.9.16 `EventSource`,
- 2.9.30 `ReplicationSink`,
- 2.9.40 `VideoStorage`, which is actually a factory for the following endpoints not published implicitly but can be looked up and accessed by `ServicePublishers`:

- 2.9.5 `ArchiveVideoSource`,
- 2.9.33 `SnapshotSource`,
- 2.9.36 `TimelineProvider`.

## 2.3 Media streaming

### 2.3.1 RTSP server

viinex 3.4 has a built-inf RTSP server for streaming video to remote clients, including other instances of viinex 3.4.

Configuration object for RTSP server in viinex 3.4 is denoted by object type `rtspsrv`. This configuration object may have four members, all of which are optional: `port`, `auth`, `transport` and `mcast_base`. An example for RTSP server configuration is given below:

```
{
    "type": "rtspsrv",
    "name": "rtspsrv0",
    "port": 1554,
    "transport": ["udp","tcp","mcast"],
    "mcast_base": ["224.0.0.70", 20000],
    "max_connections": 400
}
```

The `port` parameter defines the TCP port number the RTSP server should listen on. If not specified, the default value of 554 is used.

The `auth` section of configuration object is ignored in viinex 3.4. Instead, if authentication is required at the RTSP server, the authentication provider object should be defined in Viinex configuration, as described in section 2.7, and it should be linked with the RTSP server object in the `links` section of the configuration. In this case, authentication would be required from connecting clients. Note that viinex 3.4 only supports digest authentication for RTSP clients [13].

The `transport` parameter defines the RTP transport priority when negotiating with connecting RTSP clients. The syntax for value of that parameter is described in paragraph 2.8.1. `transport` parameter is optional; if not specified, it's value defaults to

```
"transport": ["udp","tcp"]
```

which enables both UDP unicast and TCP (in that order of preference), but disables UDP multicast. Note that if `"mcast"` value is specified as one of the elements of `transport` parameter, it is required that multicast group and base port are explicitly set in the `mcast_base` parameter.

The `mcast_base` parameter is optional, unless `transport` includes value `"mcast"` (in the latter case it becomes mandatory), and should have the syntax of a tuple of two elements, represented as JSON array of two elements. The first element of that tuple should be a string – an IPv4 address – interpreted as the address of multicast group to send the data to, if server and client

choose UDP multicast during RTP transport negotiation. Second element of the tuple should be an integer, interpreted as base port number. viinex 3.4 RTSP server, when sending data over RTP multicast, uses different port for each video source. The base port number defines the lowest port in the range to be used. The highest port number is automatically inferred from the base port number and the number of video sources served by viinex 3.4 RTSP server instance. viinex 3.4 assigns one separate multicast port number to each published live video source. For instance, given the above example, if two live video sources are published in RTSP server `rtspsrv0`, their data would be sent to multicast addresses `224.0.0.70:20000` and `224.0.0.70:20001`. The correspondence between video source and multicast address is established by viinex 3.4 automatically and sent to the RTSP client in the SDP document, so the only information required to set up multicast streaming by means of viinex 3.4 is the multicast group address and base port number.

Another optional parameter `max_connections` can be used to limit the maximum number of connections being held by the RTSP server.

To specify which video sources and/or video archives should be accessible via viinex 3.4 RTSP server, they need to be linked to the instance of RTSP server in the `links` section of the configuration document. Each video source linked to the RTSP server is published under its `name`. Therefore, given the RTSP server convifuration given above, and the following `links` section,

```
"links": {
...
   ["rtspsrv0", "cam1"],
   ["rtspsrv0", "cam2"],

   ["rtspsrv0", "stor1"],
   ["stor1", ["cam3","cam4"]],
...
},
```

video sources `cam1` and `cam2`, and video archive `stor1` would be published in RTSP server `rtspsrv0`. The videocameras, as live sources, would become accessible at RTSP URIs

`rtsp://SERVERNAME:1554/cam1`

and

`rtsp://SERVERNAME:1554/cam2`

respectively.

If a link is established between a video archive and RTSP server, the RTSP URI for obtaining of a video record is defined by the TCP port which the RTSP server listens on, the `name` parameter value of the video archive object, and the `name` parameter value of the video source object stored within the video archive:

`rtsp://SERVERNAME:`*`port/VideoArchiveName/VideoSourceName`*.

There is also possible to specify `begin=` and `end=` parameters in the RTSP URI to select the time interval in the stored video. The syntax for specifying that parameters matches that in HTTP requests for HLS stream or for exporting video (see sections 3.5.6, 3.5.5).

Given the above configuration example, it is possible to get access to video records from sources `cam3`, `cam4` in video archive `stor1` at RTSP URIs like

```
rtsp://SERVERNAME:1554/stor1/cam3?begin=1478545200000&end=1478545800000
```

or

```
rtsp://SERVERNAME:1554/stor1/cam4?begin=2017-10-08T21:00:53.231Z
                                  &end=2017-10-08T21:05:00Z
```

respectively. It is also possible to not specify the `begin=` and `end=` parameters, in which case the RTSP server would give the access to all available video for corresponding video source. In both cases, to perform navigation within an open RTSP session, the RTSP client should specify `Range:` header in his `PLAY` requests. viinex 3.4 RTSP server supports the `npt=` and `clock=` time units in `Range:` header (see [4] for more details).

Note that in order for stored video records to become available via the RTSP server, it is not required that corresponding live video sources are linked with the same RTSP server. When a video archive is published within the RTSP server, the latter publishes all video records stored in that video archive, with no connection to live video sources.

Endpoint(s) implemented by objects of type `rtspsrv`:

- 2.9.39 `VideoSourceClient`,
- 2.9.6 `ArchiveVideoSourceClient`,
- 2.9.41 `VideoStorageClient`,
- 2.9.7 `AuthnzClient`,
- 2.9.12 `DynamicVideoClient`,
- 2.9.23 `MetricsProducer`.

### 2.3.2 WebRTC server

viinex 3.4 is able to restream the live media sources to remote clients compatible with WebRTC stack of protocols (see [16], [17], [18], [19], [20], [21], [22] for more information).

For that, the object of type `webrtc` is used. This object acts as a video data consumer for media video sources in viinex 3.4, and provides an HTTP API which should be exposed via the webserver. The HTTP API is used for "signaling" purpose (according to the WebRTC jargon), — that is, by means of HTTP calls remote clients express their need for creation of a WebRTC session for a specific video source, acquire the SDP offer for respective WebRTC session, and respond with an SDP answer. HTTP API for the WebRTC object is described in section 3.14.

There are simple rules on how media of specific types and codecs is sent to the user:

- If the media source produces H.264 video, it is sent to client as is, without modifications;
- If the media source produces H.265 video, it is sent to client after decoding and subsequent encoding into H.264. The transcoding is done automatically[9], as the media is requested;

[9]There are no means to configure how the transcoding should be done. viinex 3.4 uses hardware accelerated decoder and encoder implementations, whenever possible, as described in section 2.8.3 – as if the value `auto` is provided to choose both the decoder and encoder implementation. The environment variables `VNX_HW_DECODER` and `VNX_HW_ENCODER` can be set to the value `0` to completely disable an attempt to select a hardware-accelerated codec implementation.

- If the media source has an audio track, – viinex 3.4 transcodes audio from whatever original codec into Opus, immediately upon retrieval. WebRTC server sends Opus audio to client without additional modifications.

The configuration of a `webrtc` object could look as follows:

```
{
    "type": "webrtc",
    "name": STRING,
    "ice_servers": ICESERVERS,
    "stunsrv": INTEGER,
    "ice_candidates_filter": {
        "blacklist": [STRING],
        "whitelist": [STRING]
    },
    "max_sessions": INTEGER,
    "max_sessions_per_user": INTEGER,
    "key": FILEPATH,
    "certificate": FILEPATH
}
```

There are two mandatory parameters in that configuration, `key` and `cerificate`, while other parameters are optional.

The `key` and `certificate` values should contain a path to the private key file and X.509 certificate file, respectively, which are going to be used by the WebRTC server for DTLS handshake. A sample private key and self-signed certificate[10] are installed when deploying viinex 3.4 on Windows. One can generate a new private key and a self-signed certificate using OpenSSL with the following command:

```
openssl req -newkey rsa:2048 -nodes -keyout privkey.pem -x509 \
    -days 365 -out certificate.pem
```

The `ice_servers` parameter value has the same syntax and semantics[11] as the `iceServers` parameter described in [27], section 4.2.4 – "`RTCIceServer` Dictionary".

viinex 3.4 has also a built-in STUN server component, which makes the deployment of WebRTC streaming server easier in certain scenarios. To enable the STUN server in viinex 3.4, the optional property `stunsrv` should be set into an integer number which is interpreted as UDP port number to listen on. Demo UI available for viinex 3.4 assumes that STUN server is enabled and is listening on port 3478, so it is suggested that this port number is used. When the `stunsrv` property is omitted from WebRTC configuration, the builtin STUN server is disabled.

The `ice_candidates_filter` dictionary is a means to filter out specific ICE candidates at viinex 3.4 side before sending this information to client. This dictionary may have one or both of parameters `blacklist` and `whitelist`, each represented by an array of strings, where each string should have the form of `address/mask`, – a subnet address in decimal dot-separated

[10] The use of self-signed certificate is appropriate for the purpose of WebRTC because the client receives a certificate fingerprint in the SDP offer.

[11] Previously there was a parameter `stun` to define a set of STUN servers for Viinex. While it is still supported for backwards compatibility, it is considered deprecated and may be removed in the future versions of Viinex software. The newer `ice_servers` parameter supports providing both STUN and TURN server URIs.

form, followed by a slash character and the decimal integer – the network mask length. If the `ice_candidates_filter` is omitted, it is assumed that it contains the blacklist of one element – the `169.254.0.0/16` subnet, this filtering out the IPv4 autoconfigured network interfaces.

Two optional settings `max_sessions` and `max_sessions_per_user` serve for limiting the number of sessions which can be opened – overall for this instance of the webrtc server, and by a particular authenticated user, respectively. These settings may be used to prevent denial of service because of resource exhaustion. While a webrtc session is a lightweight object, it still consumes a UDP socket and the RAM which is allocated for socket buffers. Creating a few thousand of webrtc sessions is cheap for the client and may result in exceeding the limits on open file descriptors or used RAM. It is recommended to set these settings to reasonable values in accordance to expected usage pattern.

In order to publish live video sources using the WebRTC server implemented in viinex 3.4, the former should be linked with the `webrtc` object in the `links` section of the configuration document. Also, it is required that the `webrtc` object is exposed in at least one HTTP server (the `webserver` object), so that its signaling API becomes available to remote clients. Note that despte some video sources may be published in that `webserver`, it is unrelated to the set of video sources that are going to be available via WebRTC. The latter is only affected by what video sources are linked with the `webrtc` object.

An example for configuration of object of type `webrtc` is given below:

```
{
    "type": "webrtc",
    "name": "webrtc0",
    "ice_servers": [
        {
            "urls": "stun:stun.l.google.com:19302"
        },
        {
            "urls": "turn:demo.viinex.com:3478",
            "username": "login",
            "credential": "password"
        }
    ],
    "stunsrv": 3478,
    "ice_candidates_filter": {
        "blacklist": ["10.0.0.0/8", "169.254.0.0/16"]
    },
    "max_sessions_per_user": 32,
    "key": "etc/ssl/private/sample-privkey.pem",
    "certificate": "etc/ssl/private/sample-certificate.pem"
}
```

Endpoint(s) implemented by objects of type `webrtc`:

- 2.9.39 `VideoSourceClient`,
- 2.9.6 `ArchiveVideoSourceClient`,
- 2.9.41 `VideoStorageClient`,
- 2.9.7 `AuthnzClient`,

- 2.9.12 DynamicVideoClient,
- 2.9.44 WebRTC,
- 2.9.23 MetricsProducer.

## 2.4 Media processing and auxiliary objects

### 2.4.1 Video renderer

Video renderer is the component which performs video decoding, video rendering and further video encoding, to act as a separate video source for the clients connecting to the viinex 3.4 instance. It can be used for multiple purposes, for instance to provide mobile clients with ability to view multiple live video streams (by means of performing video decoding on the server side). Another example is to embed the subtitles in the images on the video stream coming from an IP camera.

An example configuration for the video renderer component could look like

```
{
  "type": "renderer",
  "name": "rend0",
  "refresh_rate": 30,
  "share": true,
  "shm": 64,
  "transforms": [[0,
    {
      "type": "projective",
      "matrix": [
        0.6684, -1.7117,   -0.5508,
        0.0738,  1.6030,    0.1292,
        0.0198,  0.2072,    1
      ]
    }
  ], ...],
  "layout": {
    "size": [1280, 960],
    "background": "c:\\temp\\background.jpg",
    "nosignal": "c:\\temp\\nosignal.jpg",
    "viewports": [
      {
        "input": 0,
        "dst": [0.1,0.1,0.7,0.7]
      },
      {
        "input": 1,
        "border": [0,0,255],
        "dst": [0.6,0.05,0.95,0.4]
      },
      {
        "input": 2,
```

```
          "border": [0,255,0],
          "dst": [0.6,0.45,0.95,0.9]
        },
        {
          "border": [0,255,0],
          "dst": [0.1,0.75,0.35,0.95]
        },
        {
          "input": 2,
          "border": [255,0,0],
          "src": [0.1,0.3,0.6,0.6],
          "dst": [0.05,0.45,0.55,0.9]
        }
      ]
    },
    "overlay": [
      {
        "left": 50,
        "top": 100,
        "colorkey": [128, 128, 128],
        "initial": "C:/temp/qq.html"
      },
      {
        "left": 350,
        "top": 400,
        "colorkey": [128, 128, 128]
      }
    ],
    "decoder": "auto",
    "encoder": {
      "type": "cpu",
      "quality": "small_size",
      "profile": "high",
      "preset": "ultrafast",
      "dynamic": true
    }
  }.
```

The `type` of the video renderer component is `renderer`. There can be four parameters in the video renderer configuration, besides the common required `type` and `name` parameters: these are `refresh_rate`, `layout`, `encoder`, `decoder` and `overlay`. All of that parameters are optional and can be left unspecified in the configuration.

As with other components configuration, the logical connections between the video renderer and other objects are specified in the `links` section of configuration.

The `refresh_rate` should be an integer value and defines the maximum frequency of target image update. The default value for this parameter is 30. The increase of that value may improve the visual appearance of moving objects on the resulting video stream, but this comes at the price of increased usage of the CPU and the increased network bandwidth when broadcasting the resulting video stream to remote clients.

The `layout` section of the video renderer configuration should have the form of

```
"layout": LAYOUT,
```

where `LAYOUT` is the JSON object which form is described in section 2.8.5.

The `overlay` parameter is optional and defines the overlays rendered over the resulting video. The syntax and semantics of this section is described in 2.8.4.

The optional parameter `share` specifies whether the rendering should be performed to a shared memory region and made available to other processes on the same host using the local transport mechanism, see section 6.2 for more detail. Note that in order to use this option, a system privilege to create the segment of shared memory (memory mapped file) is required on Windows. This privilege should either be explicitly granted to the user, or viinex 3.4 may be run with an "evevated" privileges ("Run as administrator"). When running as a service, viinex 3.4 uses the SYSTEM account by default, which already has the required privileges to create a shared memory segment. Additionally, when the `share` option is set, the shared memory apterture size can be set by means of specifying the value of parameter named `shm`. It is expected to be set to a number of megabytes of RAM to be shared by this instance of `renderer` object. Larger value of this parameter can be used to provide better throughput of the system: in that case clients connected over local transport may hold received samples for longer without getting the server (renderer) into shared memory pool starvation state. When parameter `shm` is not specified, the default value of 32 (megabytes) is assumed.

The optional `encoder` subsection of the configuration defines the video encoder parameters applied to the video encoder to compress the resulting video stream from this video renderer instance. The syntax and semantics of this section is described in 2.8.3.

Similarly, the `decoder` setting may hold one of two string values, `"auto"` or `"cpu"`, indicating which decoder implementation should be used. Value of `"cpu"` means that harware accelerated decoder implementations will not be used with this instance. Value of `"auto"` means[12] the common logic for determining which decoder implementation to use, as described in section 2.8.3. Note that environment variable `VNS_HW_DECODER` set to 0 still effectively disables the use of hardware accelerated video decoders.

Before the input video streams are rendered on the resulting surface, a geometry transformations may be applied to them. For that, an optional property `transforms` may be specified. The value of that property should be a JSON array of pairs (each encoded as an array of 2 elements), first element of which indicates an index of the input video source, while the second should be a JSON object describing the transformation that needs to be applied to that input video source. Currently only the projective transformation is supported, which, however, is the superclass for affine transformations, shifts, rotations, shears and scaling. The projective transformation is described with a JSON object having two mandatory fields: the `type` field which should have the value of string `"projective"`, and the `matrix` field which should have the value of JSON array of exactly 9 floating-point numbers. That array should represent the matrix for the projective transformation that needs to be applied. For more information on evaluating the matrix of a projective transformation please contact Viinex support team.

Note that geometry transformations, if any, are applied prior to rendering of viewports on a resulting image. The transformation for an input channel is applied once, no matter how many viewports that input channel is rendered to. Also, the coordinates that might be specified in the `src` property of the layout configuration represent the coordinates on an image after a geometry transformation was applied to it.

[12] Other values besiders "auto" may be provided, but for simplicity they are all treated in the same way, as if "auto" is specified. That means, currently one cannot select a specific harware decoder implementation for a video renderer. This may change in the future.

Video renderer implements a number of interfaces for interaction with other components implemented in viinex 3.4. Specifically, these are: encoded video source, snapshot source, overlay control, and layout control. Mentioned interfaces are used when the corresponding links between the video renderer instance and other components are established. For more information see section 2.10.

Endpoint(s) implemented by objects of type `renderer`:

- 2.9.38 `VideoSource`,
- 2.9.39 `VideoSourceClient`,
- 2.9.33 `SnapshotSource`,
- 2.9.24 `OverlayControl`,
- 2.9.19 `LayoutControl`,
- 2.9.12 `DynamicVideoClient`,
- 2.9.13 `DynamicVideoSource` (when the `dynamic` property is set),
- 2.9.23 `MetricsProducer`.

### 2.4.2 Stream switch

Stream switch provides an efficient way to multiplex several video sources into one, giving an application using viinex 3.4 the means for controlling which one of the input video streams should appear at the output. Stream switch resembles the video renderer in that the same goal can be achieved by the latter: one could just configure the video renderer to have the same input video sources as the stream switch, and control the layout of the video renderer so that only one of the input streams is displayed on the "full screen". The important difference between this approach and the approach taken in the stream switch implementation is that video renderer decodes the video streams that need to be displayed, and then encodes the resulting video back. This decoding and encoding steps typically do consume a lot of computational resources. In contrast, the stream switch object simply switches between encoded video streams, it does not have to perform the decoding and encoding, – thus the switching between video streams costs nothing to the application.

The configuration of the stream switch is minimalistic: it should have the following form:

```
{
    "type": "streamswitch",
    "name": STRING,
    "default": INT
}
```

The type of the object for stream switch is `streamswitch`.

Just like for the video renderer object, the association between the input video sources and the stream switch is managed in the `links` section of the configuration. The sorted list of video sources that are linked to a stream switch can be obtained from the latter via HTTP API call. There is also another HTTP call – a control command – which allows an application to actually switch between input video sources. An argument to that control command is a

zero-based index of the requested video source in a sorted list of input video sources linked with that instance of stream switch. For more information see section 3.12.

The parameter `default` of the configuration specifies a zero-based index of the video source that the stream switch should pass through upon startup, before the first control command is given.

As already mentioned, the stream switch acts as a video stream consumer and should be linked with input video sources in the `links` section of the configuration. On the other hand, the stream switch is a video source itself – producing the output video stream – and hence in can be used in all contexts where video sources are used. In particular, it can be streamed as a live video source via RTSP server, written to a video archive, and so on.

Endpoint(s) implemented by objects of type `streamswitch`:

- 2.9.38 `VideoSource`,
- 2.9.39 `VideoSourceClient`,
- 2.9.35 `StreamSwitchControl`,
- 2.9.37 `Updateable`,
- 2.9.12 `DynamicVideoClient`.

### 2.4.3 Rules

As an alternative for an external management of video recording via HTTP remote procedure calls to the recording controller, viinex 3.4 offers the feature for automatic recording of video streams from a group of video sources as a reaction to some event. The origin of such events is typically a video detector working on an IP video camera, or a digital input contact.

The events acquired from ONVIF devices are processed by an object of type `rule`. It filters out irrelevant events, interprets the relevant ones, and issues corresponding commands to the recording controller.

An example for configuration section of a `rule` object is given below:

```
{
    "type": "rule",
    "name": "rule1",
    "filter": ["MotionAlarm"]
}
```

Note that the sources for acquiring events, and the recording controller which should be managed by the instance of `rule` object, are configured in `links` section of the configuration document. The only property specific to the `rule` object in the above configuration section is the `filter` variable. It should contain a list of event topics, which should be interpreted by the rule as signals to start or stop a video recording.

Here are the names of event topics recognized by viinex 3.4:

```
MotionAlarm
SignalLoss
```

```
GlobalSceneChange
ImageTooDark
ImageTooBright
ImageTooBlurry
DigitalInput
```

These names correspond to the names of event topics in ONVIF Imaging [10] and Device IO [11] Services Specifications[13]. `MotionAlarm` events are issued by ONVIF devices as a reaction to motion detection. `SignalLoss` is an event specific for analog-to-IP video converters; it means that the signal from an analog CCTV camera or from other analog source was lost. Events `GlobalSceneChange` and `ImageToo{Dark|Bright|Blurry}` are issued by so-called "service detectors" and represent the detected fact of image quality degradation or image scene change which may happen if the camera was shifted, rotated or occluded. The `DigitalInput` event is raised by an ONVIF device when it detects a state change on its GPIO contacts.

The `rule` object is capable of handling events of multiple types originating from multiple sources. Each pair of an event topic and an event source is distinguished as a separate "alarm reason". For example, a `MotionAlarm` from `camera1` and `MotionAlarm` from `camera2` are obviously different alarm reasons. An event may raise an alarm or withdraw it (for instance if a motion detector reports a presence or an absence of activity in the scene). The `rule` object keeps track for active alarms and their reasons. It issues a command to begin video recording if there are some active alarms, and stops recording when there are none (i.e. all alarms were withdrawn by corresponding events).

Endpoint(s) implemented by objects of type `rule`:

- 2.9.15 `EventSink`,
- 2.9.29 `RecControlClient`.

## 2.5 General purpose objects

### 2.5.1 Modbus GPIO device

Besides integration with ONVIF-compatible devices and registering events originating from such devices, as described in section 2.1.2, viinex 3.4 supports obtaining of GPIO events and controlling of actuators connected via devices that support Modbus protocol. viinex 3.4 supports devices connected via Modbus TCP (over TCP/IP) and Modbus RTU protocols.

Respective object in viinex 3.4 has the implementation type `modbus`, and its configuration should have the form of:

```
{
    "type": "modbus",
    "name": STRING,

    "REM": "EITHER (for Modbus TCP):",
```

[13] Strictly speaking, the event topics in mentioned specifications have the form similar to `tns1:VideoSource/MotionAlarm`, `tns1:Device/Trigger/DigitalInput`, and so on. For simplicity, the repetitive parts `tns1:VideoSource/` and `tns1:Device/Trigger/` should be stripped out when corresponding event topics are specified in viinex 3.4 configuration.

```
    "host": STRING,
    "port": INT,
    "unit_id": INT,
    "period": INT,

    "REM": "OR (for Modbus RTU):",
    "device": STRING,
    "slave_id": INT,
    "baudrate": INT,
    "bits_per_word": 8 | 7 | 6 | 5,
    "stopbits": "one" | "two",
    "parity": "even" | "odd" | "none",
    "flow_control": BOOLEAN,
    "timeout": INT,

    "inputs": { "start": INT, "count": INT },
    "outputs": { "start": INT, "count": INT }
}
```

There are two mutually exclusive groups of connection-related parameters corresponding to devices supporting Modbus TCP or Modbus RTU respectively. There are also optional sections `inputs` and `outputs` describing the GPIO addresses (basically, pin numbers) to be monitored or controlled with this `modbus` object instance.

The group of parameters related to Modbus TCP is recognized by the use of parameter `host`, which, when specified, should contain an IP address or a DNS name of the Modbus TCP device or gateway. Optional parameter `port` may specify TCP port number to connect to; when not specified the default value of 502 is assumed. An optional parameter `unit_id` may hold the logical address of device (as there can be more than one device behind the Modbus TCP interface). When not specified, `unit_id` is assumed to be equal to 1. Additional parameter `period` can be used to specify how much time, in milliseconds, should elapse between poll requests from viinex 3.4 to Modbus TCP device. When not specified, this interval is assumed to be equal to 20 milliseconds.

Alternatively, the parameter `device` can be specified instead of `host`. In such case, it should hold the name of the device file which represents the serial port interface within the operating system where viinex 3.4 runs (like `/dev/ttyS0`, `/dev/ttyUSB0` on Linux or `COM1:` on Windows). If this parameter is specified, it is assumed that Modbus RTU implementation is used. Along with `device`, a number of optional parameters specified:

- `baudrate` may be provided to set the communication speed on the selected serial interface. The numeric value should be one of the following values: 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200. If not specified, baud rate of 9600 is assumed.
- `bits_per_word` should specify the number of bits in the item transmitted over serial line. Default value is 8.
- `stopbits` may specify the number of stop bits. Default value is `one`.
- `parity` may specify the semantics of the parity bit. Default value is `even`.
- `flow_control` may be used to indicate the "handshaking" method. Default value is `false` which corresponds to no handshaking; the value of `true` means software handshaking (XON/XOFF).

- `timeout` parameter indicates the amount of time, in tenths of second, which is used for waiting of the response from Modbus RTU device when such response is expected. When not specified, value of 1 (100 milliseconds) is assumed.

Besides the above parameters for setting up the communication over the serial port, parameter `slave_id` may be provided to indicate the logical address of the Modbus RTU device (analogous to `unit_id`, – this distinction in parameter names for Modbus TCP and RTU is just to follow respective Modbus protocol flavor terminology).

Parameter sections `inputs` and `outputs` describe which GPIO contacts should be monitored or controlled by this `modbus` object instance, respectively. If either of these sections is omitted, it is assumed that the object does not monitor or does not control any GPIO pins. As follows from the configuration format, viinex 3.4 only allows a range of contiguous GPIO pins to be specified for input and for output.

Upon startup the `modbus` object tries to connect to the device specified in its configuration, reads the digital input states from the device, and sends that information in form of events (one event per each digital input). After that, the `modbus` object begins to continuously monitor the state of digital inputs, and analyses the changes of that state. When a digital input state change is detected, the respective event containing the new state is sent. In other words, a `modbus` object instance acts as an event source, implementing a semantical 2.9.16 `EventSource` interface. This means that `modbus` object can be used like any other event source – linked with event consumers like HTTP server (to publish events via WebSocket protocol), and with an instances of `rules` and `scripts` to use the events from digital inputs to perform required actions according to application-specific logic.

The events generated by the `modbus` object have the form of

```
{
    "topic":"DigitalInput",
    "timestamp":TIMESTAMP,
    "origin": {
        "type":"modbus",
        "name":STRING,
        "details": { "pin": INT }
    },
    "data":{"state":BOOLEAN},
}
```

where the parameter `origin.name` contains the identifier of the `modbus` object, `origin.details.pin` contains an index of the contact (digital input) whose state has changed, and the `data.state` contains the new state of a digital input. An example of such event is

```
{
    "topic":"DigitalInput",
    "timestamp":"2019-02-18T21:26:15.0299081Z",
    "origin": {
        "type":"modbus",
        "name":"moxa0",
        "details": { "pin": 0 }
    },
    "data":{"state":true},
}
```

which means that the very first digital input pin (`"pin":0`) of a Modbus controller monitored by object `"moxa0"` has changed on February 18th 2019 at 21:26:15 UTC, and its new state became logical “1” (`"state":true`).

In addition to the events interface, the object of type `modbus` exposes the endpoint `2.9.34 Stateful`, which can be used to read the current status of monitored input pins at any time (i.e. not only when the input pin status gets changed). The output on the read from this interface may look as follows:

```
{
  "0": true,
  "1": true,
  "2": true,
  "3": true
}
```

— this example is shown for the device where configuration section `inputs` has the value `start` set to 0 and `count` set to 4.

In order to provide the means for controlling the GPIO ports, the object of type `modbus` exposes endpoint `2.9.37 Updateable`. The `update` method accepts a JSON object containing the stringified identifiers of pins to be changed as the keys of the collection and the new logical values for the pins as the values of the collection:

```
{
  "5": false,
  "7": false
}
```

— sending such command into the Updateable interface of the `modbus` object means that IO ports number 5 and 7 should be set both to logical “0”. Respectively, the `outputs` section in the configuration of such `modbus` object should contain the pins 5 and 7, for example in can be configured to have value `start` set to 4 and value `count` set to 4.

Endpoint(s) implemented by objects of type `modbus`:

- `2.9.16 EventSource`,
- `2.9.34 Stateful`,
- `2.9.37 Updateable`,
- `2.9.23 MetricsProducer`.

### 2.5.2 Relational database connection

As per July 2023, viinex 3.4 implements using relational databases the persistence for three kinds of information: these are events generated by variuos objects which are event sources, ACL-related information to be used by authentication and authorization provider, and arbitrary JSON data organized as a key-value store.

This section describes how an object which represents the interface to a relational database within viinex 3.4 can be set up.

viinex 3.4 supports PostgreSQL and SQLite for storing abovementioned types of data.

The integration with these database engines is represented by different viinex 3.4 objects, which have types of `postgres`, `sqlite` and `mssql` respectively. However the configuration for these objects have much in common. Thus, the configuration section for a database connection object may look as follows:

```
{
    "type": "postgres" | "sqlite" | "mssql",
    "name": STRING,
    "connect": {
        "host": STRING,
        "port": INT,
        "database": STRING,
        "user": STRING,
        "password": STRING
    },
    "connections": INT,
    "load": ARRAY<STRING>,
    "inline": STRING,
    "events": {
        "store": BOOLEAN,
        "writers": INT,
        "limits": {
            "max_count": INT,
            "max_depth_abs_hours": INT,
            "storage_aware": INT
        }
    },
    "acls": BOOLEAN,
    "kvstore": {
        "enable": BOOLEAN,
        "namespace": STRING
    }
}
```

The type of the object should be `postgres` or `sqlite`.

The mandatory section `connect` specifies the information required to connect to the database instance: a host name or an IP address (property `connect.host`, optional), a port number (property `connect.port`, optional), a database name to connect to (property `connect.database`), the user name to access database server (property `connect.user`), and the password (property `connect.password`, optional). The interpretation of properties within `connect` section differs across integrated database engines. See the subsections below for connection-related information on specific RDMBS engine.

All other parameters are optional as well. The property `connections` specifies how many connections viinex 3.4 is allowed to establish to database instance simultaneously (i.e. a connection pool size). This value affects the number of queries which can be ran concurrently. When this limit is reached, viinex 3.4 does not attempt to execute a new query until an older query which is being executed completes, thus freeing the connection and placting it back to the pool. The default value for connection pool size is 4, except for SQLite. For SQLite this value should be 1.

The parameter `load` represent the array of names of SQL files which might contain the DDL or DML expressions which need to be executed upon viinex 3.4 startup. These are custom DDL or DML, and the main purpose for having these files is to maintain the custom data schema required by user's application. viinex 3.4 itself does not use these DML for creating relations necessary for event logging (these DML are built into viinex 3.4 and are always being run, despite the presence or absence of the `load` parameter). The SQL files are sought for in the specific folder, which is

```
Program Files\Viinex\share\sql\$DBMS_ENGINE
```

on Windows, or

```
/usr/share/viinex/sql/$DMBS_ENGINE
```

on Linux. Here `$DBMS_ENGINE` is meant to be equal to respective object type name (that is, `postgres`, `sqlite` or `mssql`). SQL files are executed sequentially in order they appear in the `load` array.

The parameter `inline` serves for the same purpose as the parameter `load`, however the `inline` may contain the literal text of DDL or DML statements, thus preventing the need for additional SQL files. In case if both `load` and `inline` parameters are present, – the `inline` statements are executed in the last place.

Note that if an exception occurs as the result of the initial DDL/DML statements execution, viinex 3.4 treats this condition as a fatal error and refuses to start. For this reason, if the `load` or `inline` parameters are given, or if the `events` section is given (which results in an execution of additional DDL for creating the relations for logging events), – in either of these cases the instance of database engine needs to be available at the moment of viinex 3.4 startup. It's also worth noting that the DDL/DML statements are only executed once, upon viinex 3.4 startup. This means that database relations are not be healed or re-created while viinex 3.4 is running, – this would require the restart of the whole viinex 3.4 instance or of a configuration cluster where the `postgres` object is configured.

The `events` section specifies the behavior of the `postgres` object for logging of viinex 3.4 events into the database. In order for the events received by the database integration object (as the event sink) to be logged into the database, the `events.store` needs to be set to `true`. The default value for this property is `false`.

The optional `events.writers` property specifies the number of concurrent threads which perform the write operators to store the received events to the database. The default and recommended value for this property is 2 (except for SQLite).

The optional `events.queue` property specifies the maximum length of the queue of events to be written into the database. This value may affect the behavior of the system if the database instance is under heavy load, or if there is a large number of events being generated and received by `postgres` object in a unit of time. The default value for this property is 1000.

The optional `events.limits` section specifies how the event log table should be rotated. In the course of execution, as events are being generated, the size of respective table in the database may grow to high values, while the relevance of older events may be pretty low. Usually the older events should be wiped out from the database. The `events.limits` section provides three options for this: number of events can be limited based on the age of events (according the the wall clock), and this can be specified in the property `events.limits.max_depth_abs_hours`, – this property sets the maximum age of an oldest event in the database, in hours. (The value of

720 is roughly equal to maximum event age of 1 month). The property `events.limits.max_count` specifies the maximum number of events to be kept in the database. The events are sorted by their timestamp, in descending order, and the first `max_count` events are kept, while the rest are wiped out. The property `events.limits.storage_aware` means that the age of events to be wiped out is queried from the video storage(s), so that the depth of a video storage effectively limits the depth of event storage.

> **NB!** In order for the `events.limits.storage_aware` option to take its effect, the `postgres` object needs to be linked with one or more `storage` objects in the `links` section of viinex 3.4 configuration.

Now, there's an attention needs to be paid with regard to the way how the limits described in the `events.limits` section are applied. Zero or more of the limits may be specified. If no limits are specified or the `limits` section is omitted, then the event log table is not limited by viinex 3.4. If only one limit is specified, then it works exactly as described above. However if more than one limit is specified, all of them are calculated, but *the oldest limit takes effect*. This contrasts with the logic of specifying the limits for video storage.

The database integration object in viinex 3.4 implements the event sink endpoint, which means that in order to receive the events that need to be stored to the database, the `postgres` object needs to be linked with event sources. Said event sources could be ONVIF video cameras, video analytics modules, external processes, scripts, and so on.

In order to retreive the events stored to the database later by means of HTTP requests, an HTTP API is provided. That API is described in section 3.15.

In order to manipulate ACL-related information stored in the database, there is another part of HTTP API which is described in section 3.16.

Endpoint(s) implemented by objects representing a connection to relational database:

- 2.9.15 `EventSink` (if property `events.store` is set),
- 2.9.14 `EventArchive` (if property `events.store` is set),
- 2.9.41 `VideoStorageClient` (if property `events.store` is set),
- 2.9.2 `AclProvider` (if property `acls` is set),
- 2.9.3 `AclStorage` (if property `acls` is set),
- 2.9.17 `KeyValueStore` (if property `kvstore.enable` is set),
- 2.9.23 `MetricsProducer`.

The following subsections describe details of configutation for specific database engines.

### PostgreSQL integration details

The only RDBMS supported in viinex 3.4 is PostgreSQL of version 9.6 or above[14]

[14] As per March 2021, viinex 3.4 does not install PostgreSQL automatically. Instead, it is required that a user deploys the database. viinex 3.4 accesses the PostgreSQL instance using the credetials provided, and does not use any features of PostgreSQL any newer than the support for JSONB data type, which requires PostgreSQL 9.6. On Windows, viinex 3.4 comes with libpq client, and on Linux it depends on PostgreSQL client package, – and this is sufficient to communicate with virtually any version of PostgreSQL. Specifically, versions 9, 10 and 11 were tested.

The `connect` object in the configuration is amended in ways standard for PostgreSQL clients. If any of optional parameters are omitted from the `connect` section, the usual rules for PostgreSQL clients apply in order to determine the default values for these parameters. See PostgreSQL documentation for more information on this (https://www.postgresql.org/docs/9.6/libpq-envars.html). If the whole section `connect` is omitted, the default value is used for it as shown in the above example (database "viinex", user name and password "viinex", PostgreSQL instance on a localhost running on a default port of 5432).

An example of configuration for `postgres` object is given below:

```
{
    "type": "postgres",
    "name": "pg0",
    "connect": {
        "host": "localhost",
        "port": 5432,
        "database": "viinex",
        "user": "viinex",
        "password": "viinex"
    },
    "connections": 4,
    "load": ["test.sql"],
    "inline": "CREATE TABLE IF NOT EXISTS TBL(A INTEGER);",
    "events": {
        "store": true,
        "writers": 2,
        "limits": {
            "max_count": 40000,
            "max_depth_abs_hours": 720,
            "storage_aware": true
        }
    },
    "acls": true
}
```

### SQLite integration details

For SQLite database, the integration object should have the type of `sqlite`.

The `connect` section within object configuration has the same properties as for other database engines, however all of them are ignored and may be left empty, except for the property `connect.database`, which should have the value equal to a path to SQLite3 database file on a local filesystem. If the file does not exist – it will be automatically created at object startup.

The `connection` property in the configuration – should be 1 for SQLite. Same is true for property `events.writers`.

As for the rest, – the SQLite integration very much resembles that for PostgreSQL. SQLite database is a recommended choice in case when there is no requirements for online replication, backup, or no need for multiple instances of viinex 3.4 to use a single relational data storage. No maintenance is required for SQLite database, no additinal steps for installing the server software and creating the database instance, which allows for instant deployment of viinex 3.4 with database storage in many scenarios.

### 2.5.3 Script

In order to provide flexibility for using viinex 3.4 in various scenarios, the latter has built-in support for scripting.

That support is implemented just like all other objects' implementations in viinex 3.4. Namely, an object of type `script` is introduced. Objects of that type represent an instance of JavaScript engine[15]. The instances of that objects run in parallel independently of each other and do not have shared data structures (although they of course can share the same JavaScript code, completely or partially, see below). Each instance of script, being essentially a JavaScript execution context, runs in a single thread.

Scripts can serve for the following purposes in viinex 3.4:

- maintain internal state according to a custom logic, and expose a part of that state via HTTP API (see section 3.17.1);
- accept simple requests to update the internal state and reply to such requests to the parties who initiate them (see section 3.17.2);
- receive and process events from other objects linked to this `script` object; generate and send new events;
- query and control other objects linked to this `script` object, – for example, video recording controller, PTZ device, video renderer, stream switch, and so on, – by means of calling respective JavaScript methods for such objects (see chapter 4).

The configuration of the `script` object should have the following form:

```
{
    "type": "script",
    "name": STRING,
    "load": [STRING],
    "inline": STRING,
    "onload": STRING,
    "ontimeout": STRING,
    "onupdate": STRING,
    "onevent": STRING,
    "clusters": BOOLEAN,
    "init": JSON
}
```

All of above parameters are optional (except the `type` and `name` which are mandatory for every object in viinex 3.4), but there will be typically at least the `load` or `inline` values set. Below the explanation of that parameters' meaning is given.

The most important parameter is `load`, which, if given, should be an array of strings, where each string should represent a path to a file on a local filesystem containing JavaScript source code. All files specified in the `load` parameter should be present and readable at the time of viinex 3.4 startup, otherwise the `script` object reports configuration error. The files mentioned in this parameter are executed sequentially, in the order they are mentioned.

[15] To be precise, viinex 3.4 uses the Duktape engine https://duktape.org/ which implements ECMAScript 5 [23] as per November 2018.

The parameter `inline` may contain an inline JavaScript code, – that is, some part of code can be included directly in viinex 3.4 configuration. If both `load` and `inline` parameters are present, – the JavaScript code contained in the `inline` parameter is executed after all code from files mentioned in the `load` array have run.

The source code from the files mentioned in the `load` array, as well as the source code from the `inline` parameter is executed only once, when viinex 3.4 instance is being started (or when the cluster is being created).

It is important that at the time when that code is executed, the `script` object is not yet linked to any of viinex 3.4 objects (and they might be not created by the time). None of API specific to viinex 3.4 is available at the time when `load` and `inline` code is executed. That's why it is recommended that this code only contains definition for functions and/or data structures but does not try to perform any actions for side effects.

Another option to load the script code is to organize that code as JS modules, place them in a predefined paths which viinex 3.4 uses to search for JS files when executing the `require()` function (see section 4.2.6), and load as a normal module. After that, however, the `onload`, `ontimeout`, `onupdate` and `onevent` functions (see below) that might be defined by that module, need to be lifted into the root naming context of the script, – and for that there is a pre-authored module named `vnx-script-instance`. Thus, a script configuration to use the JS implementation code from a module could look as follows:

```
{
    "type": "script",
    ...
    "inline": "require('vnx-script-instance')('script-impl.js');",
    ...
}
```

where `script-impl.js` is a substitute for an actual script implementation module name.

Four parameters `onload`, `ontimeout`, `onupdate` and `onevent` may contain the names of JavaScript functions that should be called to handle respective *events* (here *event* means not a viinex 3.4 event, but a certain point in the lifecycle of the JavaScript state machine). These handlers have the following meaning:

- `onload` handler is called when the script code is completely loaded, and all viinex 3.4 objects are linked. Basically, this is an entry point, when the instance of script can complete its initialization, having the access to viinex 3.4 API and linked objects, and begin its normal operation.
- `ontimeout` handler is called when (and if) the timer, previously established by this script for itself, rings.
- `onupdate` handler is called when a request for the `Updateable` interface HTTP call described in section 3.17.2 is received from the HTTP server where this `script` object is published.
- `onevent` handler is called when an event is received from one of event sources that this instance of `script` is linked with.

For more information on the use of that handlers in scripting see chapter 4. By default, if some of that four parameters are omitted, it is assumed that the respective handler should

have the name exactly matching the parameter name (that is – `onload` handler has the name `onload`, and so on). Alternatively, the handler names can be overriden. The handlers with either default or overriden names should be the top-level JavaScript functions.

An optional boolean parameter `clusters`, when set to `true`, gives the respective `script` an access to the functionality of managing viinex 3.4 configuration clusters, as described in section 4.2.8. When omitted, the value of this property is assumed to be `false`.

The parameter `init` may take an arbitrary JSON value which is passed as an argument into the `onload` handler when the latter is called. This is a mechanism for passing the initialization parameters into the script, which can be convenient if the source code of the script is written to be reusable: properly written reusable script can be made applicable in all scenarios it was designed for without the need for altering its source code for every use case: the actual adjustment of the script behaviour can be performed from the viinex 3.4 configuration by means of changing the value of `init` parameter.

It is important that in the runtime each instance of the `script` can only get the events, update requests, publish its state, and interact in any other possible way only with those viinex 3.4 objects that are linked with this instance of `script` in the `links` section of the configuration. The viinex 3.4 objects that are not linked with the script are inaccessible for it.

Each instance of the `script` object in viinex 3.4 implement the internal interfaces of event source (that is – a script can be used in all links where an event source is required), event consumer (that is, it can be linked with event sources), stateful, and updateable (see section 3.17).

For more information on scripting in viinex 3.4 please refer to chapter 4.

Endpoint(s) implemented by objects of type `script`:

- 2.9.32 `ServicePublisher`,
- 2.9.16 `EventSource`
- 2.9.15 `EventSink`,
- 2.9.37 `Updateable`,
- 2.9.34 `Stateful`.

### 2.5.4 External process

#### Overview

Besides the scripting capability, viinex 3.4 allows one more way of integration with third party systems – by means of interfacing using an external process.

The idea behind this resembles the FastCGI technique for interfacing between web servers and various related applications. The webserver could start an external process, manage its lifecycle, and communicate with that process via named pipes or UNIX domain sockets. Likewise, viinex 3.4 may start an external process, manage its lifecycle, and communicate with that process using appropriate IPC methods.

An object for starting and interacting with an external process should have the type `process` in viinex 3.4 configuration.

There are two mechanisms provided for communication between viinex 3.4 and an external process, which serve for two different purposes:

- Events interchange. An external process can obtain events from viinex 3.4, and that process can also generate events and pass them to other objects in viinex 3.4. For that, the simple mechanism of standard input/output is used. viinex 3.4 starts the external process with pipes connected to that process' input and output file descriptors (0 and 1 respectively). All events that the `process` object is subscribed to via `links` section of the configuration – are serialized into JSON format and sent by viinex 3.4 to the standard input of that process. And vice versa, everything that the process writes to its standard output is expected to have JSON syntax, and if it conforms, – it is accepted by viinex 3.4 as an event from that process. If there are event consumers linked with that instance of `process` object, they get events produced in this way.
- Video processing. viinex 3.4 provides a native API to obtain uncompressed video from raw video sources or from video renderers. Using that API, an external process can obtain the video frames, for instance, to perform some custom video analytics. The IPC methods that are used to implement the API to acquire the uncompressed video stream in viinex 3.4 are shared memory and named pipes (on Windows) or UNIX domain sockets (on Linux), so the raw video interchange between processes on the local machine is performed with zero copying. For some more information on the native API see section 6.2.

These two mechanisms enable a number of applications. An event-based integrations with third-party systems can be made relatively easy in almost any programming language, having in mind the simple interface for events interchange (that is, the standard input/output of data in JSON format). The native interface for raw video processing requires about half a dozen of native function C calls, which also should not be a problem in most programming environments; in exchange the application efficiently gets the live raw video stream ready for analytics. It's up to the application what to do with the results of that analytics – it can be either injected back to viinex 3.4 in form of events, or it can be stored or passed for further processing using some independent method.

### Configuration

The configuration of an object of type `process` should have the following form:

```
{
    "type": "process",
    "name": STRING,
    "cmdline": STRING |
    "executable": STRING,
    "args": [STRING],
    "cwd": STRING,
    "env": [ [STRING, STRING] ],
    "restart": BOOLEAN,
    "timeout": BOOLEAN,
    "init": JSON
}
```

Here, the only mandatory parameters, besides the `type` and `name`, are `cmdline` or `executable` (mutually exclusive). All other parameters are optional.

The `cmdline` parameter specifies the command line to be executed in order to start an external process. If given, this command is executed by means of the shell (command interpreter). As an alternative, the parameter `executable` can be given in order to specify the path to an executable file that needs to be run. If that option is given, the parameter `args` might be useful to set the command line argument for running that executable. The value of this parameter should be a JSON array of strings, – each string representing a single command line argument, as they would have been split by the shell (separated with spaces). If the `args` parameter is not given, it is assumed that no command line arguments should be passed when running the process.

> **NB!** Setting the `cmdline` parameter is only supported in viinex 3.4 for Linux. On Windows the only available way of specifying how the process should be started is setting the path to process' executable with the `executable` parameter, and optionally setting the command line arguments via the `args` array.

The optional `cwd` parameter allows the working directory of the newly created process to be set. If this parameter is omitted, the newly run process inherits the working directory from the viinex 3.4 instance that the process is supervised by.

The `env` parameter should have the form of array of pairs of strings (each of which is, in its turn, encoded as a JSON array of 2 elements), and provides the way to set the environment variables set for the newly created process. If this data is given, each pair in the top-level array `env` is interpreted as the pair – an environment variable name (the first element of a tuple), and an environment variable value (the second element of a tuple). The environment given by the `env` paramener is merged with the environment inherited by the newly created process from its supervising viinex 3.4 instance; wherein the variables provided in the `env` array override those from inherited environment.

The lifecycle of the process is managed by viinex 3.4. This means that the process is started by viinex 3.4, and the latter watches for the status of the process. If the process stops, viinex 3.4 has an option to either leave it stopped, or (typically) to restart it. This behaviour is defined by the `restart` parameter. If not set, the default vaule is assumed to be `true`, – that is, the failed process is restarted by default.

Upon shutdown, viinex 3.4 tries to shut down its child processes gracefully. In order to do that, viinex 3.4 closes the input and output file descriptors associated with the external process. The implementation of the process should read its standard input, and, when the EOF (end of file) is received from the STDIN file descriptor, – the external process should complete its activity as soon as possible and quit.

If an external process breaks this protocol, and fails to complete after its standard input was closed, – viinex 3.4 would forcefully terminate such process. The parameter `timeout` defines the time interval, in seconds, that viinex 3.4 should wait after closing external process' STDIN, before terminating that process. The default value for this parameter is 10 seconds.

Last but not least, the `init` configuration parameter is intended for passing an arbitrary initialization information into the external process. This parameter can take a value of any JSON type and form. When the process is started, the very first message it receives on the standard input is this initialization value, serialized into JSON. The rest of JSON values (second, third, and so on) coming to process' standard input are the events coming from viinex 3.4. This is true even if the `init` value is not set: in this case, the first JSON value coming to the external process' standard input would be `null`.

### Runtime protocol

Like it was mentioned above, the external process receives at its standard input at least one JSON value, which is equal to the value of `init` property of that `process` object configuration. After that, the process receives the events from event sources that it is linked to in the `links` section of viinex 3.4 configuration.

Implementations should read out the data from their standard input stream file descriptor, and wait for the end of file signal on that descriptor. This EOF signal on the stdin must be interpreted by implementations as the command to shut down. After the EOF is received on the stdin, the external process should complete its activity and exit as soon as possible.

The strings that are written by external process to its standard output are interpreted by viinex 3.4 as JSON records, in order to generate events and send them to event consumers linked with that instance of the `process`. Upon receival, each of these JSON records is examined for the presence of two fields, `topic` and `data`. The `topic` field is mandatory, it should be present, and its value should have the type `STRING`. This field serves to set the topic of the event to be generated. The field `data` is optional and serves to set the payload of the event to be generated. The resulting event is formed by viinex 3.4 to match the following pattern:

```
{
    "topic": STRING,
    "timestamp": TIMESTAMP,
    "origin": {
        "type": "Script",
        "name": STRING
    },
    "data": VALUE
}
```

where `topic` and `data` are substituted from the JSON record received from the external process, while the `timestamp` is set to the current time (in UTC timezone), and `origin.name` is filled with the identifier of this `process` object, according to viinex 3.4 configuration.

Another IPC channel provided between viinex 3.4 and an external process is the standard error stream of the process, which serves for the logging purpose. All strings written by the process to its file descriptor 2, which is standard error stream, are caught by viinex 3.4 and directed into its own log (syslog or log file) on behalf of respective `process` object. The severity level assigned to such log messages is determined from first character of the line sent by the external process into its file descriptor 2:

- `D` stands for `DEBUG`,
- `I` stands for `INFO`,
- `W` stands for `WARNING`,
- `E` stands for `ERROR`.

Upon retrieval of the line from process' file descriptor 2, viinex 3.4 determines the log level of the message, chops off the first two characters, and logs respective message with the level which was defined by the process. This way, messages sent by external process into its file descriptor 2, will appear in viinex 3.4 log.

As stated in the overview, implementations are free to use the functionality for obtaining raw live video streams from the instance of viinex 3.4 they are supervised by, using the API briefly described in chapter 6.

Endpoint(s) implemented by objects of type `process`:

- 2.9.16 `EventSource`,
- 2.9.15 `EventSink`.

### 2.5.5 Web server

viinex 3.4 has a built-in web server implemented for making media data accessible, and to expose API for applications using viinex 3.4.

Configuration object for web server in viinex 3.4 is denoted by object type `webserver`. Such configuration object may contain two optional fields: `port` and `staticpath`. An example for video archive configuration is given below:

```
{
  "type": "webserver",
  "name": "web0",
  "port": 8880,
  "staticpath": "/usr/share/viinex/html",
  "static": [
    ["images","/usr/share/images"],
    ["logs","/var/log"]
  ],
  "tls": {
    "key": "etc/ssl/private/sample-privkey.pem",
    "certificate": "etc/ssl/private/sample-certificate.pem",
    "chain": ["etc/ssl/private/ca1.pem", "etc/ssl/private/ca2.pem"],
    "allowed_versions": ["1.3", "1.2", "1.1"]
  },
  "cors": "*",
  "hls": {
    "fragments": 3,
    "duration": 5
  },
  "clusters": true
}
```

The field `port` defines a TCP port number which viinex 3.4 HTTP server should listen on. Default value for this field is 8880; this port number is chosen if `port` field is absent in HTTP server configuration. The `staticpath` field should contain path to a folder on local filesystem where viinex 3.4 static web content is deployed. The contents of specified folder is served by viinex 3.4 web server starting at root URL.

The `auth` section of configuration object is ignored in viinex 3.4. Instead, if authentication is required at the HTTP server, the authentication provider object should be defined in Viinex configuration, as described in section 2.7, and it should be linked with the RTSP server object in the `links` section of the configuration.

The optional parameter `tls`, if present, instruct the instance of the web server to act as HTTPS server rather than a plain HTTP server. Note that each instance of `webserver` object can only be either HTTP or HTTPS server, not both at the same time. The `tls` property, when present, should be a JSON object containing two mandatory properties, `key` and `certificate`, specifying the path to the private key and certificate, respectively, on a local filesystem. The key and certificate files should be in PEM format and the key should not be protected by password. There may also be an optional parameter `tls.chain` to specify the chain of intermediate certificates. The `chain` property, if present, should be a JSON array of strings, each pointing to a local PEM file containing a certificate of an intermediate certification authority. An optional parameter `tls.allowed_versions` may be used to specify the TLS versions to be supported by this web server instance. When set, this property should have the type of array of strings, and the possible string values are: `"1.3", "1.2", "1.1", "1.0"` for TLS versions 1.3–1.0 respectively, and also `"3"` and `"2"` for SSL versions 3 and 2 (not recommended for actual use). If the parameter `tls.allowed_versions` is not set, only the TLS versions 1.3 and 1.2 are enabled.

The optional parameter `cors` is intended to set up the cross-origin resource sharing policy for the web server instance. This parameter can take a value of string or of an array of strings, each denoting a separate `Origin` for which cross-origin resource sharing is declared to be permitted. If the `cors` parameter is set to an array of strings, there can be several such origins; if it is set to a single string, that string defines the only allowed origin. There are two special cases for `cors` parameter: this is a single string containing an asterisk, `"*"`, which denotes that the instance of web server being configured permits cross-origin connections from arbitrary origins, – and an empty string, which is equivalent to not setting the `cors` parameter at all, and denotes that the web server instance does not support CORS. This is also the default behaviour.

The optional parameter `hls` allows for tuning how the video streams are published via HLS in the web server instance. This parameter should be a JSON object with two integer properties, `fragments` and `duration`. The former defines the number of fragments published in HLS playlist for each live video source within the web server. The latter defines the minimum duration for each fragment. Note that actual fragments' duration depends on video source's encoder settings. When publishing a video stream via HLS, viinex 3.4 splits transport stream into fragments on GOP boundaries. Therefore, if `duration` value is set to 1 second, but the I-frames in published video stream come no more frequent than once in every 5 seconds, – the actual fragment size would be 5 seconds for that video stream. If the `hls` parameter substructure is omitted, the default settings are applied, which instruct viinex 3.4 to prepare for each published live video source a playlist of 3 fragments, each of at least 5 seconds long.

The `clusters` parameter, when set to `true`, makes the cluster-related API available under that instance of webserver. The cluster-related API is described in section 3.19 of this configuration. Note that this includes the cluster management itself (creation, removal, enumeration of existing clusters) as well as accessing the objects published by means of dynamic clusters, and receiving the events from such objects via the WebSocket interface. All of that functionality is enabled by the `"clusters": true` flag. By default, if not set explicitly, the clusters-related functionality is disabled for the specific webserver instance, as if the `"clusters"` parameter was set to `false`.

Endpoint(s) implemented by objects of type `webserver`:

- 2.9.32 `ServicePublisher`,
- 2.9.12 `DynamicVideoClient`,
- 2.9.15 `EventSink`,

- 2.9.7 AuthnzClient,
- 2.9.9 ClusterManagerReader,
- 2.9.11 ConfigReader,
- 2.9.23 MetricsProducer.

### 2.5.6 Publisher for objects in configuration clusters

Publishing of objects created in the main static configuration of viinex 3.4 instance requires to link that objects with an instance of `webserver` object. However, when it comes to publishing the objects created dynamically within a cluster, the problem is that objects within a cluster can only "see" and interact with each other but not with objects in other clusters and/or main (static) configuration. As a consequence, objects in a dynamically created cluster can only be linked with a webserver(s) created in the same cluster, which is typically not what is required, because that webservers would require a new HTTP port, different from the HTTP port of the webserver in the main configuration.

The solution is to have a formal object of type `"publish"` which can be described in the configuration of a cluster and used to link other objects in that cluster. Establishing such link means that each object in a cluster linked to a `"publish"` object becomes available in the HTTP server(s) in the main configuration via HTTP API calls described in 3.19.6. This behavior has no options to change, so all the instances of a `"publisher"` object within a cluster are essentially equivalent, and therefore there is no reason to have more than one instance of such object in a cluster. The configuration of this object is shown below:

```
        {
            "type": "publish",
            "name": "publish0"
        }
```

where the only variable parameter is the name of that object (here `"publish0"`, and it only plays the role for authoring the `"links"` section of cluster's configuration. A simple example for cluster's configuration could look like

```
{
    "objects":
    [
        {
            "type": "onvif",
            "name": "cam1",
            "host": "192.168.0.131",
            "auth": ["admin","admin"]
        },
        {
            "type": "publish",
            "name": "publish0"
        }
    ],
    "links":
    [
```

```
        ["cam1", "publish0"]
    ]
}
```

— here, an ONVIF device is created with a name `"cam1"`, and its live video stream is published in the cluster-enabled webserver (which should have been created in the static part of viinex 3.4 configuration). If a cluster with name `cluster1` is created with the configuration given above, the video stream from ONVIF camera with address 192.168.0.131 is going to be available at URI `http://SERVER:PORT/v1/cluster/cluster1/cam1/stream.m3u8`. For more information on clusters-related HTTP API see section 3.19.

For the purpose of section 2.10, the `"webserver"` and the `"publish"` concepts are interchangeable. Establishing a link between a publisher and some specific object is semantically equivalent to establishing a link between a web server and that object.

Endpoint(s) implemented by objects of type `publish`:

- 2.9.32 `ServicePublisher`,
- 2.9.12 `DynamicVideoClient`,
- 2.9.15 `EventSink`,
- 2.9.7 `AuthnzClient`,
- 2.9.11 `ConfigReader`.

### 2.5.7 WAMP client

viinex 3.4 implements an experimental[16] support for the Web Application Messaging Protocol, WAMP [26]. WAMP provides means for remote procedure calls as well as event distribution in a pub-sub model. It is important that application components are not accessing each other directly during WAMP interaction; rather they talk to a middleware component called WAMP router. The specification suggests that one of the transport-level protocols for WAMP could be a WebSocket. The typical WAMP router then would be a WebSocket server, while applications are going to connect to this server, acting like a WebSocket clients. After connection is established, however, client's role is not limited with how it was established; each client can potentially make remote procedure calls, accept and process such calls, subscribe for events, as well as publish events.

Within viinex 3.4 the support for WAMP is implemented by object of type `wamp`, whose configuration should have the form as follows:

```
{
    "type": "wamp",
    "name": STRING,
    "realm": STRING,
    "auth": {
        "method": "cryptosign",
        "role": STRING,
        "secret": HEXSTRING
```

[16] Experimental here means that certain aspects of behavior are likely to change.

```
    },
    "url": STRING,
    "app": STRING,
    "prefix": STRING,
    "clusters": BOOLEAN
}
```

Besides the common `name` and `type`, the parameters `realm`, `url` and `prefix` are mandatory, while the rest are optional. The `url` should be a URL for connecting to a WebSocket server, i.e. it should start with `ws:` or `wss:` schema, contain a host name or an IP address, optionally followed by a port number, and a path. These parameters depend on the WAMP router which viinex 3.4 needs to connect to; please refer to documentation for respective router. The `realm` parameter defines the isolated scope (realm in terms of WAMP) to connect to. The `app` and `prefix` parameters, together, define the prefix prefix URI for event topics and procedure names published by this instance of WAMP client on the router. The resulting prefix is formed as the values of `app` and `prefix` parameters, concatenated and separated with a dot character. The resulting prefix should be a WAMP URI, as defined by section 5.1.1 of specification [26]. If the optional `app` parameter is omitted, the default value of `com.viinex.api` is taken. The mandatory `prefix` part should be so that resulting prefix is unique for each object of type `wamp` within a WAMP realm.

The optional configuration parameter group `auth` serves for the purpose of enabling the authentication when WAMP client connects to the router. The only authentication method supported by viinex 3.4, besides anonymous connection, is Cryptosign [28], based on Ed25519 asymmetric encryption algorithm. The parameter `auth` should be a JSON dictionary, with key `method` set to a string value `"cryptosign"`, key `role` set to the name of role/user (this can be also called `authid` in the documentation of Crossbar router), and the key `secret` should be a string of hexadecimal digits, representing Ed25519 keypair (64 bytes of data, 128 hex digits) or a "seed" to build the keypair (32 bytes of data, that is 64 hex digits), which is equivalent to the private (leading) part of the keypair.

The optional parameter `clusters` specifies whether this instance of `wamp` client should expose the functionality for managing viinex 3.4 clusters over the RPC interface. When omitted, the value of `false` is used, and the clusters-related part of API is not published.

Besides these configuration parameters, the object of type `wamp` may be linked with authentication and authorization provider (`authnz`). When such link is established, the access to API exposed by the instance of `wamp` object needs to be authorized, and before that the accessing principal needs to be authenticated. This happens within exactly the same logic which is descibed in section 2.7.

When linked with other viinex 3.4 objects, the WAMP client exposes the API to control them, and, in case if some of these objects emit events, – publishes the events.

> **NB!** Currently, if the object of type `wamp` is linked with object of type `authnz`, – no events are published over WAMP protocol.

An example of configuration of object of type `wamp` is given below:

```
{
    "type": "wamp",
    "name": "wamp0",
    "realm": "realm1",
```

```
    "auth": {
        "method": "cryptosign",
        "role": "viinex",
        "secret": "56416b7c0ed7a2e6...2e54829d3d4e22f1"
    },
    "url": "ws://127.0.0.1:8080/ws",
    "app": "com.viinex.api",
    "prefix": "wamp0",
    "clusters": true
}
```

Here the object named `wamp0` is created to connect to a WebSocket server running at localhost on port 8080, and it's going to work in WAMP realm ``realm1''[17]. The remote procedures exposed by this object will start with prefix

```
com.viinex.api.wamp0,
```

and same it true for events published by this object. For more details on how API and events are published please refer to chapter 5.

Endpoint(s) implemented by objects of type `wamp`:

- 2.9.32 `ServicePublisher`,
- 2.9.15 `EventSink`,
- 2.9.7 `AuthnzClient`,
- 2.9.11 `ConfigReader`,
- 2.9.9 `ClusterManagerReader` (if property `clusters` is set),
- 2.9.23 `MetricsProducer`.

### 2.5.8 Floating license server

viinex 3.4 requires a license document for functioning, and the license document could be bound to a USB device (SenseLock dongle), to the server's hardware (identidied by so called `hwid`), or to some network interface card on the server (identified by the NIC's MAC address). However these parts may be hard to manage if some kind of containers (Docker, LXDE) or virtualization is being used for deployment. This could also be the case in a cloud environment, where an instance may be migrated from one physical server to another.

For this reason, viinex 3.4 supports the model of licensing when a pre-defined set of viinex 3.4 components may be ran on several physical or virtual hosts, and still use the same shared license document. Such licenses may be considered floating across these hosts. viinex 3.4 instances each running on these hosts would use the license leases issued based on the same license document, and this is performed with the help of a floating license server.

The configuration for such server looks as follows:

[17] These settings match that established by the Crossbar.io WAMP router implementation https://crossbar.io/, when installed with default settings.

```
{
    "type": "licmgr",
    "name": "licmgr0"
}
```

so it basically has no parameters to set except the name of license server object. The object type for license server should be `licmgr`.

The license server object should be linked with one or more web servers, as described in section 2.10. The web server may impose its own requirements on authentication and/or the use of TLS. The floating license server exposes an `Updateable` programming interface described in section 3.17.2. The specific format of updates acceptable by the floating license server is not disclosed; it is private between the floating license server and floating license client. However the mechanism for this API exposure is common for all components that implement the `Updateable` API. What is important is that the floating license server having the name `licmgr0`, when linked with a web server, is published under the URL `http://SERVERNAME:PORT/v1/svc/licmgr0`, and this URL should be specified in the configuration of floating license clients wishing to access respective floating license server. The configuration of a floating license client is described in section 2.12.2.

The floating license server dynamically uses the licenses (leases) from the license document which is specified at its hosting viinex 3.4 instance. When a floating license client requests a license lease from the server, the latter acquires such lease locally on behalf of that client. When the client returns the lease, respective licenses get returned to the local pool of licenses. In this way all floating license clients connected to the same license server share the same license pool, and the same pool is used for viinex 3.4 objects that might be co-located with floating license manager (on the same viinex 3.4 instance).

The interaction between license server and its client is designed to be fault-tolerant, i.e. the license lease do not get invalidated when a network fault occurs. As a rule, after a short network outage the leases are successfully renewed for both client and server. Even if a longer network outage occur, the client "survives" the absence of connection to license server for up to 8 hours. The server keeps the expired leases taken by a client as long as there is no lack for licenses. Only in that case the expired licenses are considered "released" by the server. However even in that case, and also in the case if the license server gets restarted, – the floating license clients continue to work, and attempt to re-acquire the license leases they need. The fault only occurs in two cases: when the license client is working for more than 8 hours without the ability to contact the license server, or if after it contacts the license server, when while attempting to re-acquire the license leases that are currently in use, the client faces server's explicit refusal to acquire required license leases. In this case license manager client shuts down its viinex 3.4 instance.

Endpoint(s) implemented by objects of type `licmgr`:

- `2.9.37 Updateable`,
- `2.9.34 Stateful`.

Floating license server exposes `2.9.37 Updateable` endpoint for use by floating license clients. The exact protocol is internal, subject to change and is not disclosed.

Endpoint `2.9.34 Stateful` can be used on a floating license server to find out the number of licenses currently in use by the clients of this server. Here's an example of the call to `2.9.34 Stateful` endpoint on a floating license server:

```
$ curl -s -X GET https://localhost:8889/v1/svc/licmgr0 -k | jq
[
  [
    "ViinexCore",
    1
  ],
  [
    "IpVideochannel",
    8
  ]
]
```

### 2.5.9 Metrics

Some of the objects within viinex 3.4 expose the endpoint 2.9.23 `MetricsProducer`, which means that these objects gather statistical information about themselves. In order to conveniently provide access to that information, the object of type `metrics` can be used. This object serves for two puposes: a) it accumulates all the metrics produced by `MetricsProducers` linked to this object, and b) when linked to a 2.9.32 `ServicePublisher`, which is an HTTP server or a WAMP client, – this object provides the API for the serialized metrics data to be scraped and stored for further use.

The configuration of the object of type `metrics` may look as follows:

```
{
  "type": "metrics",
  "name": STRING,
  "labels": [ [STRING, STRING] ],
  "runtime": BOOLEAN
}
```

Here the optional parameter `labels` can hold the array of pairs of strings (each pair encoded as a JSON array of exactly two elements), which define common labels added to every metric collected and published by this instance of `metrics` object. Typically these should be the labels to help with identlifying viinex 3.4 instance and/or cluster where the information was collected.

The optional parameter `runtime` tells whether this instance of `metrics` object should capture and publish the metrics from Haskell runtime – these can be used to monitor the overall performance of viinex 3.4 instance. When not set, it's assumed that this property is set to `true`.

The metrics are provided in the Text-based Prometheus Exposition Format [30], in order to be compatible with Prometheus https://prometheus.io/ and Victoria Metrics https://victoriametrics.com/, so that both of these can be set up to scrape the data directly from viinex 3.4. For instance, if an object of type `metrics` has the name `metrics0` and is created in the main viinex 3.4 cluster, and linked with an object of type `webserver` which listens at port 8880, – then Prometheus can be configured to scrape the metrics from HTTP endpoint `http://hostname:8880/v1/svc/metrics0`.

If the object of type `metrics` is linked with a WAMP client, the API would look differently. Within WAMP, the interface `MetricsExporter` exposes one method named `sample`, which

takes no parameters and returns the metrics data as a string, GZipped and then base64-encoded. It would be client's responsibility to decode and decompress the result of this method call, but when it's done – the result can be pushed into Prometheus push gateway or into Victoria Metrics' endpoint `/api/v1/import/prometheus`, as described in https://docs.victoriametrics.com/#how-to-import-data-in-prometheus-exposition-format.

Note that the object of type `metrics` only collects the metrics data from other metrics producers (and the runtime system, unless the flag `runtime` is set to `false`), so it needs to be explicitly linked with those which are of interest for analyzing their load, performance, etc. The amount and coverage of the statistical data depends on which viinex 3.4 objects are linked with the metrics consumer. Only the objects which implement the endpoint 2.9.23 `MetricsProducer` can be instrumented.

Endpoint(s) implemented by objects of type `metrics`:

- 2.9.21 `MetricsConsumer`,
- 2.9.22 `MetricsExporter`.

## 2.6 Third-party video management systems

This section describes the configuration options for viinex 3.4 objects which represent connection points to specific third-party video management systems. That objects should be linked to `vmschan` objects in order to give the latter specific "flavour" (i.e. to actually associate a brand-neutral `vmschan` object with a video channel at the specified VMS instance). For more information on `vmschan` object configuration see section 2.1.5

All product and company names mentioned throughout this section are trademarks or registered trademarks of their respective holders. Use of that product and company names does not imply any affiliation with or endorsement by any of them.

The objects described in this section implement the same two endpoints:

- 2.9.42 `VmsConnection`,
- 2.9.34 `Stateful`.

The `VmsConnection` endpoint is required to link the VMS object with its respective `vmschan`nels. The `Stateful` endpoint can be used to publish it on a HTTP API or read it within an embedded script. However the syntax of JSON data provided by each particular VMS over the `Stateful` endpoint is not subject of this documentation; it may depend on how the VMS integration is implemented and which data this integration may provide.

### 2.6.1 Milestone XProtect

For interaction with Milestone XProtect VMS product family (https://www.milestonesys.com/solutions/platform/video-management-software/), an object of type `milestone` needs to be created in viinex 3.4 configuration. It should have the following form:

```
{
    "type": "milestone",
```

```
    "name": STRING,
    "host": STRING,
    "port": INT,
    "auth": [STRING,STRING],
    "certificate": STRING
}
```

In the above, property `host` should contain the IP address or a resolvable host name of the server where the instance of Milestone XProtect Management Server can be reached. Optional property `port` should contain the TCP port which is used by Milestone XProtect Management server to expose its Management API. If not specified, the value of port number 443, which is the default for C-Code product family, is assumed.

A pair of strings under the property `auth` should contain the login and password which should be used by viinex 3.4 to authenticate at Milestone XProtect VMS instance.

> **NB!** In the context of interoperation with Milestone XProtect, viinex 3.4 implements only Basic HTTP authentication in conjunction with TLS, when accessing Milestone XProtect Management Server API. NTLM authentication is not supported for that purpose. For that reason, login and password that viinex 3.4 uses to authenticate at XProtect Management Server should not be login and password of a Windows user. The credentials specified in `auth` section of `milestone` object configuration, should appear in "Security – Basic Users" section in Milestone XProtect Management Client.

Optional property `certificate` may contain the path to a local file holding the TLS certificate, in PEM format, that is used by the instance of Milestone XProtect Management server. If `certificate` property is omitted, viinex 3.4 would connect to XProtect Management server using TLS, but without checking the certificate authenticity. This scenario should be avoided in public networks.

Milestone XProtect integration in viinex 3.4 supports specifying a video stream using either the camera identifier (GUID), or the camera human readable name. For that, properties `channel.id` or `channel.name` of `vmschan` object configuration should be used, respectively:

```
    "channel": {"id":STRING}
    "channel": {"name":STRING}
```

The human readable name of camera can be found in its configuration properties in Milestone XProtect Management Client. For that,in the Management Client, in section "Devices – Cameras", a camera should be selected. Its name would appear in the top of camera's property sheet. To obtain the unique identifier of a camera, the camera should be selected in Management Client with Control key pressed on the keyboard. With Control key pressed, the same camera's property sheet would appear, but in the bottom of that page a readonly text area should be visible, containing the records "ID=(GUID)" (and probably other values). For more information on finding camera GUID in Milestone XProtect please refer to the following article in Milestone KBase: https://supportcommunity.milestonesys.com/s/article/finding-camera-GUID. Camera GUID value obtained in this way can be used in viinex 3.4 configuration, in the video channel selector property `channel.id` of respective `vmschan` object.

An example of viinex 3.4 object for integration with Milestone XProtect is given below:

```
{
    "type": "milestone",
    "name": "xp1",
    "host": "192.168.0.123",
    "auth": ["Admin","12345"]
}
```

### 2.6.2 Geutebrück G-Core

An integration object for G-Core VMS by Geutebrück GmbH (https://www.geutebrueck.com/) is represented in viinex 3.4 by a configuration in the following format:

```
{
    "type": "geutebrueck",
    "name": STRING,
    "host": STRING,
    "auth": [STRING, STRING]
}
```

The `host` and `auth` parameters are both mandatory. Parameter `host` should hold the value of IP address or name of the server where G-Core instance is running. The `auth` parameter should represent a pair of strings – the user name and password to connect to G-Core instance.

> **NB!** Geutebrück does not disclose the network protocol for interaction with G-Core server. The only way for third parties to interact with G-Core is by means of C++/C# SDK provided by vendor, and in case of Geutebrück this SDK is only provided for Windows. For that reason, G-Core integration is only available in Windows builds of viinex 3.4.

G-Core integration in viinex 3.4 supports specifying a video stream using either the "Media channel ID" (in the terminology of G-Core), media channel's "Global number" (an integer number), or the human readable name of the channel. For that, properties `channel.id`, `channel.global_number` or `channel.name` of `vmschan` object configuration should be used, respectively:

```
"channel": {"id":STRING}
"channel": {"global_number":NUMBER}
"channel": {"name":STRING}
```

Although "Media channel ID" can be seen in G-Set interface[18], Geutebrück documentation advises against using this value to permanently identify the camera. The value of "Media channel ID" is used to identify the camera during the session, throughout all G-Core SDK API calls, however G-Core documentation says this value may change over time, in contrast with "global number" (which can only be changed explicitly by a user). To sum up, – the recommended camera identification method for G-Core intergration in viinex 3.4 is the identification by means of `global_number` parameter.

An example of viinex 3.4 object for integration with Geutebrück G-Core is given below:

[18]The "Media channel ID" of camera in G-Core is an integer number, as well as the "Global number" identifier. For consistence with other VMS integrations, the actual type of `channel.id` parameter is expected to be string. For G-Core integration this string is converted to integer number by the G-Core integration plugin.

```
{
    "type": "geutebrueck",
    "name": "gcore1",
    "host": "192.168.0.102",
    "auth": ["sysadmin","masterkey"]
}
```

### 2.6.3 Qognify (SeeTec) Cayuga

An integration object for Cayuga VMS by Qognify GmbH (former SeeTec), https://www.qognify.com/products/cayuga/, is represented in viinex 3.4 by a configuration in the following format:

```
{
    "type": "cayuga",
    "name": STRING,
    "host": STRING,
    "auth": [STRING, STRING],
    "port": INT,
    "certificate": STRING
}
```

> **NB!** The integration of Cayuga VMS implementation in viinex 3.4 requires that two server-side components are installed and enabled at Cayuga server: the SeeTec Gateway Service and the Transcoding Service. For that, respective items should be checked when installing Cayuga software.
> The Transcoding service should be allowed for use on respective Device Manager instance. For that, one should go in the Cayuga administrative UI to the Settings – Server – Transcoding Module property page, and turn on the "Assigned DeviceManager server" checkbox for respective server.

The `host` and `auth` parameters are both mandatory. Parameter `host` should hold the value of IP address or name of the server where the instance of Cayga VMS is running. The `auth` parameter should represent a pair of strings – the user name and password to connect to the Cayuga server instance.

The `port` property of configuration is optional; it may contain the values of port number for accessing the SOAP endpoint of the Cayuga server. If not specified, the `port` value is assumed to be of the value 62000, which is default for this VMS.

The `certificate` property is optional as well and may serve to specify the path to a file containing a TLS certificate which should be used to verify the authenticity of the server. The certificate file should be in the PEM format. If this property is not specified, the authenticity of the server is not checked.

In order to obtain the certificate from a local server, one may use the following command:

```
openssl s_client -showcerts -connect 127.0.0.1:62000 </dev/null | \
openssl x509 > cayuga-certificate.pem
```

The resulting `cayuga-certificate.pem` then can be copied to the host where viinex 3.4 is running.

Cayuga integration in viinex 3.4 supports specifying a video stream using the name of the camera and the human-readable number of the video source. For specifying the name of the camera, the property `channel.name` of `vmschan` object configuration should be used:

```
"channel": {"name":STRING}
```

For convenience, a selector specified by setting of a string-typed `id` is treated for Cayuga integration exactly in the same way as the one with the `name` being set; but the `id` semantics lets the user to specify just the string identifier of a camera as the value for the `channel` property:

```
"channel": STRING
```

For specifying a human-readable name of video source, the stream selector of one of the following forms should be used:

```
"channel": {"global_number": INTEGER}
"channel": INTEGER
```

The human-readable numbers can be assigned in Cayuga Client: for that, one needs to go to the Configuration Mode, on the right panes select the Company, in the Company pane select the item “System”, and in the “System” pane select the “Entity numbering”. As a result, a user interface for assigning human-readable numbers to video sources and other entities would appear.

An example of viinex 3.4 object for integration with the Cayuga VMS is given below:

```
{
    "type": "cayuga",
    "name": "vms1",
    "host": "win10-cayuga",
    "auth": ["administrator","pass"]
}
```

### 2.6.4 Pelco VideoXpert

An integration object for Pelco VideoXpert[19] is represented in viinex 3.4 by a configuration in the following format:

```
{
    "type": "pelco",
    "name": STRING,
    "host": STRING,
    "auth": [STRING, STRING],
    "port": INT,
    "certificate": STRING
}
```

[19] https://www.pelco.com/products/video-management-solution-videoxpert/vxpro/

The `host` and `auth` parameters are both mandatory. Parameter `host` should hold the value of IP address or name of the server where the instance of Pelco VideoXpert VMS is running. The `auth` parameter should represent a pair of strings – the user name and password to connect to the VideoXpert server instance.

The `port` property of configuration is optional; it may contain the values of port number for accessing the HTTP RPC endpoint of the Pelco VideoXpert server. If not specified, the `port` value is assumed to be of the value 443, which is default for this VMS.

The `certificate` property is optional as well and may serve to specify the path to a file containing a TLS certificate which should be used to verify the authenticity of the server. The certificate file should be in the PEM format. If this property is not specified, the authenticity of the server is not checked.

Pelco VideoXpert integration in viinex 3.4 supports specifying a video stream using the name of the camera and the human-readable number of the video source, and the UUID of the video source. For specifying the name of the camera, the property `channel.name` of `vmschan` object configuration should be used:

```
"channel": {"name":STRING}
```

For specifying a human-readable number of video source, the stream selector of one of the following forms should be used:

```
"channel": {"global_number": INTEGER}
"channel": INTEGER
```

The human-readable numbers can be assigned in Pelco VideoXpert: for that, one needs to go to the VxOpsCenter, on the right panes select the Content panel, in the Content panel select the tab “Sources”, and in the table on that tab each item can be edited (via the context menu), to set the “Number value” in the “Edit source” dialog box. This number value can be used to identify the respective connected Pelco VideoXpert VMS channel in viinex 3.4 configuration.

An example of viinex 3.4 object for integration with the Pelco VideoXpert VMS is given below:

```
{
    "type": "pelco",
    "name": "vms1",
    "host": "test-pelco",
    "auth": ["admin","Admin12345"]
}
```

### 2.6.5 Bosch BVMS

An integration object for Bosch BVMS[20] is represented in viinex 3.4 by a configuration in the following format:

```
{
    "type": "bosch",
```

[20] https://www.boschsecurity.com/xc/en/solutions/management-software/bvms/

```
    "name": STRING,
    "host": STRING,
    "password": STRING |
    "auth": [STRING, STRING],
    "port": INT,
    "certificate": STRING,
    "port_rtsp": INT,
    "auth_rtsp": [STRING, STRING],
    "timezone": INT | STRING,
    "direct": BOOLEAN,
    "auth_direct": [STRING, STRING]
}
```

The only mandatory parameter here is `host` which should specify an IP address or a resolvable name of the server where an instance of BVMS is running. Note that viinex 3.4 integrates Bosch BVMS in terms of "VRM instances", i.e. each Video Recording Manager should be described in viinex 3.4 configuration as a separate VMS instance.

Besides the `host` parameter, it is required that either the property `password` is specified, or the property `auth` and, optionally, also the `auth_rtsp` is specified. This set of properties works as described below:

- if only the property `password` is specified, then the credentials used for communication with BVMS are – username "srvadmin" and the given password for RCP+ over CGI communication, and username "user" and the given password for RTSP communication. This works with the default setup.
- if the property `rtsp_auth` is specified (a pair of strings representing a user name and a password), then these credentials are used for RTSP communication. Other credentials are not used for RTSP in that case.
- if the property `auth` is specified (a pair of strings representing a user name and a password), then these credentials are used for RCP+ over CGI communication with BVMS. Other credentials are not used for RCP+ in that case.
- if the property `auth` is specified, and neither `password` nor `auth_rtsp` are specified, then the password from this `auth` property is also used for RTSP interaction, however with that password the user name "user" is used in RTSP protocol.

To keep the configuration simple, it is recommended that either only the property `password` is specified (and user names are substituted implicitly), or both the properties `auth` and `auth_rtsp` are specified with explicitly mentioned user names.

The `port` and `port_rtsp` properties of configuration are both optional; each of them may contain the value of port number for accessing the HTTPS endpoint for RCP+ over CGI interaction with the VRM instance, and the port number for accessing that VRM instance over RTSP protocol, respectively. If either of that properties is unspecified, the `port` value is assumed to be of the value 443, and `port_rtsp` is assumed to be of value 554, which are the default values for a Video Recording Manager in the Bosch BVMS.

The `certificate` property is optional as well and may serve to specify the path to a file containing the TLS certificate which should be used to verify the authenticity of the server when communicating via RCP+ over CGI over HTTPS. The certificate file should be in the PEM format. If this property is not specified, the authenticity of the server is not checked.

An optional property `timezone` may contain either an integer number of minutes – the offset of timezone of the server where Bosch BVMS instance is hosted from the UTC timezone, – or it can contain a string in format ±`HHMM` to specify timezone offset from UTC in hours and minutes. The following symbolic names for timezones are accepted as well: `"UTC", "UT", "GMT", "EST", "EDT", "CST", "CDT", "MST", "MDT", "PST", "PDT"`[21]. If the `timezone` property is omitted, it is assumed that viinex 3.4 instance and Bosch BVMS instance are running in the same time zone. If these timezones actually differ, then it is important that viinex 3.4 instance knows what timezone BVMS uses; otherwise the video archive playback requests and the video archive search requests with an interval specified would produce incorrect results.

The optional property `direct`, when set to `true`, instructs viinex 3.4 to pull live video streams directly from cameras rather than from the Bosch VMS server (VRM). When this option is set, viinex 3.4 determines whether it is possible to get the stream from camera directly (for that, the VRM should expose the camera's address; generally this means that this has to be a Bosch camera rather than an ONVIF-compatible device connected via ONVIF gateway), and if so, then in order to retreive a live video stream, this address is used instead of the address of Bosch server. Coupled with the option `direct`, goes the `auth_direct` property which may contain the credentials which should be used for such direct connection to cameras. If this property is omitted and the property `direct` is still set to `true`, the same rules as for `auth_rtsp` are applied to infer the credentials for direct connection to Bosch cameras. When `direct` option is omitted, its value is assumed to be `false`.

Bosch BVMS integration in viinex 3.4 supports specifying a video stream using the name of the camera and the human-readable number of the video source. For specifying the name of the camera, the property `channel.name` of `vmschan` object configuration should be used:

```
"channel": {"name":STRING}
```

For specifying a numerical identifier of video source, the stream selector of one of the following forms should be used:

```
"channel": {"global_number": INTEGER}
"channel": INTEGER
```

For Bosch cameras, the property `channel.stream` may also be set. When the `channel.stream` is set and the property `direct` is set for the object of type `bosch`, viinex 3.4 acquires the first "video instance" of the camera as the ``main'' stream, and the second "video instance" of the camera as the ``sub'' stream.

If a camera is set up to encode video in H.265, it is necessary to specify the `hint` property of the channel selector, setting that property to string value `"h265"`:

```
"channel": {"global_number": INTEGER, "hint": "h265" }
```

An example of viinex 3.4 object for integration with the Bosch BVMS is given below:

[21] These are all U.S. timezones. It's currently impossible to properly specify a timezone with daylight savings outside of U.S. in Viinex config, but luckily this is not required for Bosch BVMS integration. Instead, a correct time offset from the UTC timezone in standard time ("winter" time, in the northern hemisphere) needs to be specified, because Bosch BVMS accepts and produces the temporal values measured as the number of seconds since January 1st, 2000 in its local time zone – which means that the offset from UTC timezone as per this date only needs to be known to perform conversion between BVMS timestamps and UTC time, conventional throughout viinex 3.4 API.

```
{
    "type": "bosch",
    "name": "vms1",
    "host": "win10-bosch",
    "password": "!Admin12345"
}
```

### 2.6.6 DSSL Trassir

An integration object for DSSL Trassir VMS (https://trassir.com/) is represented in viinex 3.4 by a configuration in the following format:

```
{
    "type": "trassir",
    "name": STRING,
    "host": STRING,
    "port": INT,
    "auth": [STRING,STRING],
    "certificate": STRING
}
```

In the above, property `host` should contain the IP address or a resolvable host name of the server where Trassir instance is being run. Property `port` should contain the TCP port which is used by Trassir to expose its API. Pair of strings under the property `auth` should contain the login and password which should be used by viinex 3.4 to authenticate at the Trassir instance. Optional property `certificate` may contain the path to a local file holding the TLS certificate of the Trassir instance. If `certificate` property is omitted, viinex 3.4 would connect to Trassir without checking the certificate. This scenario should be avoided in public networks.

Trassir integration in viinex 3.4 supports specifying a stream, along with id or name of a video channel, for the purpose of video source identification in `vmschan` objects configuration. For that, the `stream` property may be added to the value of `channel` field of `vmschan` object configuration:

```
    "channel": {"id":STRING, "stream":"main" | "sub"}
    "channel": {"name":STRING, "stream":"main" | "sub"}
```

Only the `id` and `name` are supported for video source identification within Trassir instance. The global numberic identifiers are not present for video sources in Trassir VMS.

An example of Trassir object configuration is given below:

```
{
    "type": "trassir",
    "name": "trassir1",
    "host": "192.168.0.123",
    "port": 8080,
    "auth": ["Admin","12345"],
    "certificate": "c:\\temp\\trassir1.crt"
}
```

### 2.6.7 Macroscop and Eocortex

An integration object for Macroscop[22] VMS is represented in viinex 3.4 by a configuration in the following format:

```
{
    "type": "macroscop",
    "name": STRING,
    "host": STRING,
    "auth": [STRING, STRING],
    "port": INT,
    "certificate": STRING
}
```

The `host` and `auth` parameters are both mandatory. Parameter `host` should hold the value of IP address or name of the server where the instance of Macroscop VMS is running. The `auth` parameter should represent a pair of strings – the user name and password to connect to the Macroscop server instance.

The `port` property of configuration is optional; it may contain the values of port number for accessing the HTTP RPC endpoint of the Macroscop server. If not specified, the `port` value is assumed to be of the value 8080, which is default for this VMS.

The `certificate` property is optional as well and may serve to specify the path to a file containing a TLS certificate which should be used to verify the authenticity of the server. The certificate file should be in the PEM format. If this property is not specified, the authenticity of the server is not checked.

Macroscop integration in viinex 3.4 supports specifying a video stream using the name of the camera and the human-readable number of the video source, and the UUID of the video source. For specifying the name of the camera, the property `channel.name` of `vmschan` object configuration should be used:

```
    "channel": {"name":STRING}
```

For specifying a UUID of video source, the stream selector of one of the following forms should be used:

```
    "channel": {"id": STRING}
    "channel": STRING
```

where STRING should contain the UUID of the camera. In addition, in both forms the `"stream"` property may be used which, as usual, may contain string value `"main"` or `"sub"`.

An example of viinex 3.4 object for integration with the Macroscop VMS is given below:

```
{
    "type": "macroscop",
    "name": "vms1",
    "host": "test-macroscop",
    "auth": ["root",""]
}
```

[22] https://macroscop.com/produkty/programma-dlya-ip-kamer. One may find that the same integration is also compatible with a VMS which is distributed under brand Eocortex, https://eocortex.com/.

Note that Macroscop VMS does not provide much of a network addressing information via its HTTP API, and for that reason it's not possible to determine which endpoints and ports should be used to connect to Macroscop instances in a federated setup. As a consequence, for a federated Macroscop system with multiple servers it is required that every such server is descibed in viinex 3.4 configuration, and the `vmschan` objects should be properly linked to their respective VMS instances. In other words, despite the Macroscop configuration may be a federation of instances, – these instances should be treated like isolated Macroscop instances for the purpose of configuring them in viinex 3.4.

### 2.6.8 ITV|AxxonSoft Intellect

An integration object for ITV|AxxonSoft Intellect VMS (https://www.itv.ru/products/intellect/) is represented in viinex 3.4 by a configuration in the following format:

```
{
    "type": "intellect",
    "name": STRING,
    "host": STRING,
    "port": INT,
    "timezone": STRING | INT
}
```

The property `host` should contain the IP address or a resolvable host name of the server where Intellect instance is being run. Property `port` is optional and may contain the number of TCP port of `video.run` server module. If not specified, the value of 20900 of port number is assumed. Optional property `timezone` may contain either an integer number of minutes – the offset of timezone of the server where Intellect is hosted from the UTC timezone, – or it can contain a string in format ±HHMM to specify timezone offset from UTC in hours and minutes. The following symbolic names for timezones are accepted as well: `"UTC"`, `"UT"`, `"GMT"`, `"EST"`, `"EDT"`, `"CST"`, `"CDT"`, `"MST"`, `"MDT"`, `"PST"`, `"PDT"`. If the `timezone` property is omitted, it is assumed that viinex 3.4 instance and Intellect instance are running in the same time zone.

Note that Intellect integration in viinex 3.4 does not support identification of video channels being linked (`vmschan` object) in any other way except by the `global_number`. In other words, the objects of type `vmschan` linked with `intellect` should have the configuration property `channel` set to an integer number – the numberic identifier of respective "camera" object in Intellect configuration.

An example of Intellect object configuration is given below:

```
{
    "type": "intellect",
    "name": "int1",
    "host": "192.168.0.117"
}
```

### 2.6.9 Native plugins for other VMS integrations

Besides the video management systems mentioned in this section, viinex 3.4 is open for integration of other VMS which can be added by independent parties. For that, the mechanism

of loadable plugins is being used, similar to the one for H264 video sources described in section 2.1.3. As the matter of fact, the API for H264 video source plugin is a part of the API for integration of video management systems. For more information on that API please refer to section 6.4 of this document.

The configuration for the VMS whose integration is available in such plugin should have the following form:

```
{
    "type": "vmsplugin",
    "name": STRING,
    "library": STRING,
    "factory": STRING,
    "init": JSON
}
```

The mandatory `library` and `factory` properties instruct viinex 3.4 to dynamically load a specific DLL (shared object on Linux), and find the specific symbol in it (the one with name specified in the `factory` property).

The `init` property is an arbitrary JSON value which is passed into the VMS integration plugin factory as is. It should contain the information necessary for the plugin to connect to the VMS instance: the address of a server, port numbers, credentials, and so on. The contents of this value depends on the specific VMS.

As an example, the Geutebrück G-Core integration is actually implemented as a plugin, and can be used by the following config section, as an equivalent of the example given in subsection 2.6.2:

```
{
    "type": "vmsplugin",
    "name": "gcore1",
    "library": "vmsplugins.dll",
    "factory": "create_geutebrueck_vmsplugin",
    "init": {
        "host": "test-gcore",
        "auth": ["sysadmin", "masterkey"]
    }
}
```

Here, the `vmsplugins.dll` is the name of a DLL actually shipped with viinex 3.4 distribution on Windows, which contains the implementation of VMS integration plugin for G-Core. Respectively, `create_geutebrueck_vmsplugin` is the name of the factory function exported from the `vmsplugins.dll` to create the VMS plugin instance for G-Core. Compare the content of `init` object with the example configuration for G-Core shown in subsection 2.6.2: it specifies the same configuration keys and values, in this case `host` and `auth`.

After the plugin is implemented and the respective object is created in viinex 3.4 configuration, the rest of the logic remains the same as for other VMS integrations: in order to access a specific video channel from viinex 3.4, the `vmschan` object should be created, specifying the channel selector information, and that `vmschan` object should then be linked with the `vmsplugin` object in the `links` section of viinex 3.4 configuration document.

### 2.6.10 Script-driven VMS integrations

Aside from plugins mechanism for integration of third party video management systems described in section 2.6.9, viinex 3.4 also supports so called script driven VMS integrations. The idea behind it is to have a specific `script` object (see section 2.5.3 for a general information on viinex 3.4 builtin scripts) which controls the behavior of the integration: it provides the RTSP URLs to connect to a third party systems to retreive video data; it may provide URLs to get the still images from the VMS being integrated, and it can also construct URLs to get the timeline information, and may further parse the result of respective HTTP requests to return the timeline in form recognizeable by viinex 3.4.

The script-driven VMS is represented in viinex 3.4 by object of type `scdrvms`. This object has no configuration parameters, except for mandatory `type` and `name`, and serves for the purpose to have the anchor to establish links in the `links` section. Conventionally, like all other VMS objects, this `scdrvms` object needs to be linked with objects of type `vmschan`, so that the latter receive the particular implementation. However in contrast with other VMS integrations, the `scdrvms` object also needs to be linked with an object of type `script`[23], and that script should implement certain specific protocol in its `onupdate()` method implementation.

Not that since `scdrvms` object has no configuration, the connection parameters like address information, credentials, and so on, are typically defined in the configuration of driver script object, specifically – in its `init` section.

An example of configuration for using a script-driven integration of Dahua NVR with 2 cameras is given below (the conventional parts of configuration containing the web server, WebRTC server, and so on, are omitted, – only parts specific to script-driven integration are retained):

```
{
    "objects":
    [
        {
            "type": "scdrvms",
            "name": "vms1"
        },
        {
            "type": "script",
            "name": "script0",
            "inline": "require('vms-driver')('vms-dahua');",
            "init": {
         "auth": ["admin","admin"],
                "host": "10.0.0.42"
            }
        },
        {
            "type": "vmschan",
            "name": "cam1",
            "channel": 12,
            "enable": ["video"],
            "dynamic": true
        },
        {
            "type": "vmschan",
```

[23] To be precise, it needs to be linked with an object which provides the endpoint `2.9.37 Updateable`.

```
            "name": "cam2",
            "channel": 13,
            "enable": ["video"],
            "dynamic": true
        }
    ],
    "links":
    [
        ["vms1","script0"],
        [["cam1","cam2"],"vms1"],
        [["cam1","cam2"],["web0","webrtc0","rtspsrv0"]]
    ]
}
```

The `inline` configuration property for object `script0` (the driver script) contains the construct of `"require('vms-driver')('vms-dahua');"`, which instructs viinex 3.4 to load the JS module `'vms-driver'`, – a generic loader module for VMS drivers, – which in its turn should load the specific driver module `'vms-dahua'`. As per November 2021, three VMS driver modules implemented in JS are shipped with viinex 3.4, – these are `'vms-hikvision'` which implements the driver for Hikvision NVRs; `'vms-dahua'` implementing the driver for Dahua NVRs, and the demo driver `'vms-viinex'` which could be a simple illustration of how a VMS driver may be implemented, – this one is capable of connecting remote instances of Viinex as a third party VMS into local instance of viinex 3.4.

The VMS driver script should implement certain protocol, namely, it should accept certain `onupdate()` requests with the arguments of specific forms, and should reply with results of specific forms expected by viinex 3.4. This convention is described in detail in section 4.4.

## 2.7 Authentication and authorization

### 2.7.1 Role-based access control

As of version 3.1, Viinex implements role-based access control, and implements an authentication & authorization provider object which can be used by other viinex 3.4 objects which provide connecting users with access to functionality.

The approach taken for implementing authentication and authorization assumes that most of objects' implementations within viinex 3.4 do not have to know how and by whom their functionality is being used. However there is a limited number of "gateway" object types which provide external interface for an instance of viinex 3.4. Such objects are always aware of who is accessing them, and they can perform the checks to ensure that a user is properly authenticated, and he has the permissions to access the functionality of underlying viinex 3.4 objects.

Such "gateway" object types are: HTTP server (`webserver`), its proxy object inside of a viinex 3.4 configuration cluster (`publish`), an RTSP server (`rtspsrv`), and WebRTC server (`webrtc`). Not coincidentally these and only these objects expose the `AuthnzClient` endpoints. The endpoints of respective type can be matched by the dual `AuthnzProvider` endpoint, and in case it happens, – respective "gateway" object uses the linked authentication & authorization provider object to authenticate connecting users and to check their permissions. Note that the criteria for a "gateway" object (a HTTP or RTSP server) to require users' authentication and

authorization is when this object is linked with an `AuthnzProvider`. If such link is specified, – the authentication will be required for respective network server and permissions will be controlled. If the link with an `authnz` object is not present, – the network server object would grant anonymous access to all other objects that are published in it.

Authentication and authorization provider sticks to a role-based access control model, which in viinex 3.4 implementation means that a user (principal), once authenticated, is associated with a list of roles. The decision on whether a principal may access certain resource is based solely on principal's roles, – not on its specific identity.

Now, the resources on which authorization decision should be made, are actually viinex 3.4 objects and their endpoints. This is the least granularity at which access rights may be differentiated within viinex 3.4. When an operation is requested by a principal, it is determined on which object the operation is requested, and which endpoint(s), in viinex 3.4 terminology, should be involved to fullfill the operation. Thus, the endpoint is treated as a whole, and no further division (for instance up to specific API methods within an interface which could be exposed along with endpoint) are made.

The information on permissions within viinex 3.4 is represented as tuples (triplets) of: role identifier, object identifier, and endpoint type identifier. Tuples of such kind are called access control entries (ACE) in viinex 3.4, and the zero or one or any number of ACEs form so called access control list (ACL). For brevity the special forms of ACEs are supported, where object identifier, or endpoint type identifier, or both, can be omitted (or have the `null` value). By convention in viinex 3.4 this means that any object (if object identifier part is omitted), or any endpoint (if endpoint part is omitted) is matched by such ACE.

For example, the ACE

```
["guest", "cam1", "VideoSource"]
```

provides access to the endpoint of type `VideoSource` on object with identifier `cam1` to a principal having the role ``guest''. Further, the ACE like

```
["viewer", null, "VideoSource"]
```

grants members of role ``viewer'' with access to live video source on all objects; the ACEs of the form

```
["group1", "cam_1_3", null],
["group1", "cam_1_5", null],
```

provides access to all functionality on objects with identifiers `cam_1_3` and `cam_1_5` to the members of `group1`, while the following access control entry

```
["admin", null, null]
```

provides that the principals having the role `admin` have all access to all objects (published within a server which is linked to an `AuthnzProvider` with such an ACE among others.

No other processing or interpretation are made on access control entries: they are either matched strictly, or using a `null` values which is interpreted as a whildcard for respective element of a tuple. If an access is requested for `(role, object id, endpoint type)`, but no matching entry was found on an active ACL, – then such operation is not authorized.

There are no default elements in ACLs, which means that all access to all functionality on all objects is initially denied. Note that there are no "deny" access control entries in viinex 3.4. To allow the access, an ACL needs to be populated either within a static configuration of an authentication & authorization provider object, or via the underlying database storage (see sections 2.5.2 and 3.16 for more information on this).

### 2.7.2 Token authentication

Depending on the "gateway" object which needs to check user's authenticity and permissions, it could be the case that an underlying transport protocol allows for using multiple non-persistent connections from a principal. In order to not require the principal to authenticate with every new connection or request, viinex 3.4 implements the token-based authentication. As per August 2021, this is implemented in HTTP server within viinex 3.4. Upon the first successful password-based authentication, the HTTP server requests the authentication provider object to issue a temporary token, which is passed to the principal and is used by the latter to authenticate further requests to HTTP server.

The token is signed by the issuer (an authentication provider object) and holds the information on principal's roles. This, on one hand, allows the server to bypass password-based authentication for every request, and there is no need to lookup principal's roles for every request.

As of Viinex version 3.1, the authentication token format is switched to JWT [25]. The algorithm for signing the authentication tokens issued by viinex 3.4 is HMAC SHA-256 (HS256). This implies that there is a secret to compute the HMAC. While by default a random string is chosen at viinex 3.4 startup as such secret, – as an option it can be specified in `AuthnzProvider`'s configuration. This makes it possible to reuse the authentication token issued on one viinex 3.4 instance to be used at another viinex 3.4 instance which shares the same token.

### 2.7.3 Authentication and authorization provider object

The authentication and authorization provider object in viinex 3.4 has the type of `authnz`, and its configuration should have the form of

```
{
    "type": "authnz",
    "name": STRING,
    "acl_id": INT,
    "realm": STRING,
    "secret": BASE64_STRING,
    "token_ttl": INT,
    "cache_ttl": INT,
    "htdigest": FILENAME,
    "accounts": ARRAY,
    "roles": ARRAY,
    "acl": ARRAY
}
```

Besides the standard `name` property, only the `acl_id` and `realm` properties are mandatory. Other properties have reasonable default values and may be omitted. Note however that in

order for `authnz` object to function properly, it needs a related data to be provided. That data may be provided either in properties `htdigest` and `accounts` (for credentials), `roles` (for user–roles mapping), and `acl` (for access control list), or in a database object which should implement the endpoint `2.9.2 AclProvider` and be linked to this instance of `authnz` object. Both the static configuration and the database link may be specified; in such case the information given in configuration has priority over database (for crenentials database, and is being searched first for matching ACLs). This can be useful to provide maintenance access to the system.

The `acl_id` is necessary if a database object being used as a storage for ACL infromation. The `acl_id` makes it possible to use the same database for storing independent access control lists. This should be an integer value.

The `realm` string is a value used to verify the client's response in digest authentication protocol. Some transport protocols (for example RTSP) assume this value is known in advance, before a user made a first authentication attempt. The realm value used for computing password digest is also stored in `htdigest` files and in viinex 3.4 credentials database. For this reason, where possible, viinex 3.4 attempts to use the realm which is present in the credentials database, to form the challenge for a specific user. However this is not possible for RTSP server, and thus it is advised that the `realm` value specified in this configuration match the realm which is used to compute the users' password digests when credentials database is being populated.

The `secret` parameter, if specified, should be a base64-encoded string. A decoded value can be an arbitrary string which is used to sign and verify the authentication tokens. If this parameter is not specified, a random secret is chosen. This invalidates previously issued authentication tokens, and makes different instances of `authnz` object incompatible with each other (from perspective of authentication tokens issued). If the `secret` is specified, then authentication tokens issued by one instance of `authnz` object would be accepted by all other instances of `authnz` objects sharing the same `secret` value.

A connected to the above is the value of `token_ttl`. If specified, it should be an integer value, a number of seconds for which the authentication tokens are issued. If not specified, the default value of $1800 = 30 \times 60$, i.e. 30 minutes, is used. Note that authentication is not re-issued automatically by the server. A client may examine the token, namely its `exp` claim [25], in order to estimate whether the token is still valid or a password authentication is required.

If the `authnz` object is linked with a database which stores the ACL-related information and exposes the `2.9.2 AclProvider` endpoint, then caching of such information is performed. For this purpose, the `cache_ttl` controls when cache invalidation should occur (this should be an integer number, in seconds). If not specified, a default value of 60 (1 minute) is used.

The other 4 parameters specify the static part of ACL-related information.

#### `accounts` and digest authentication

The `accounts` parameter is a simple credentials database and should have the form of JSON array of objects. Each object in that array should have one of two following forms:

```
{
  "type": "apikey",
  "key": "STRING",
  "secret": "STRING"
}
```

or

```
{
  "type": "password",
  "login": "STRING",
  "realm": "STRING",
  "digest": "STRING"
}.
```

The first form, with `"type":"apikey"`, stores the account name (in member `"key"`) and its respective plain-text password (in member `"secret"`). This is unsafe way of storing credentials, and it is intended for use when client is a trusted programmatic component (an agent, a robot), – not a human.

The second form, with `"type":"password"`, stores the account name (in member `"login"`) and the "digest" of account password (in member `"digest"`). The "digest" part is an MD5 hash of account name, the `realm` and account password, all separated by semicolon character ':', — according to the protocol for Digest Access Authentication described in [13]:

$$HA1 = MD5(login : realm : password).$$

For instance, one may compute the digest for account named `user` and password `12345` with realm `ViinexAuth` using the following UNIX command line:

```
$ echo -n "user:ViinexAuth:12345" | md5sum
e488a5401e549bcb46e59da2d065d433 *-

$ echo -n "guest:ViinexAuth:54321" | md5sum
6e23b775d34d9c1119779ce57d6d47e7 *-
```

This allows the server for checking of user's credentials via HTTP and RTSP protocol without knowing the account's plain text password.

#### `htdigest`

The `htdigest` property, when set, should point to a locally accessible text file produced by command-line utility `htdigest(1)` usually coming with `apache2-utils` package. The file formed by `htdigest` utility is a text file consisted of lines, where each row describes one account's credentials. The description includes account name, the realm and the digest, computed as described above. For instance, one may issue the following commands:

```
$ htdigest -c ./htdigest.txt ViinexAuth user
Adding password for user in realm ViinexAuth.
New password: 12345
Re-type new password: 12345

$ htdigest ./htdigest.txt ViinexAuth guest
Adding user guest in realm ViinexAuth
New password: 54321
Re-type new password: 54321
```

```
$ cat ./htdigest.txt
user:ViinexAuth:e488a5401e549bcb46e59da2d065d433
guest:ViinexAuth:6e23b775d34d9c1119779ce57d6d47e7
```

Using the `htdigest` option in viinex 3.4 configuration is a convenient way of avoiding the need for changing the configuration files when a user changes his password. Additionaly, the `htdigest` utility is de-facto standard and is supported on many platforms, including Windows and Linux.

If `htdigest` parameter is provided, it should point to an existing file of correct format. Failure to read and parse specified file will result in an error at viinex 3.4 startup. viinex 3.4 also does not recognize the changes to htdigest file while running: in order for such changes to take effect, the viinex 3.4 instance should be restarted.

Once read, the entries found in the `htdigest` file are stored in memory, and can be used to authenticate, along with information found in relational database.

#### `roles`

The `roles` property, when set, should be a JSON array of pairs, each represented as JSON array of exactly 2 elements. The pupose of this parameter is to establish a static user–role mapping. Thus, in most basic case, the `roles` array may consist of pairs of strings. First element in each pair represents the user name (login), while the second element – role name.

For brevity, the Cartesian product notation may be used to associate multiple users with the same role, or assign multiple roles to one users, or to assign same set of multiple roles to a set of multiple users. In particular, the following notiations are (pairwise) equivalent:

```
[ ... , ["user", ["role1", "role2", ..., "roleM"]] , ... ]
  <=> (is equivalent to)
[ ... ,
  ["user","role1"],
  ["user","role2"],
  ...
  ["user","roleM"],
 ... ],
```

as well as

```
[ ... , [["user1", "user2", ..., "userN"], "role"] , ... ]
  <=> (is equivalent to)
[ ... ,
  ["user1","role"],
  ["user2","role"],
  ...
  ["userN","role"],
 ... ],
```

and

```
[ ... , [["user1", "user2", ..., "userN"],
```

```
        ["role1", "role2", ..., "roleM"]] , ... ]
  <=> (is equivalent to)
[ ... ,
  ["user1","role1"],["user1","role1"],...,["user1","roleM"],
  ["user2","role1"],["user2","role1"],...,["user2","roleM"],
  ...
  ["userN","role1"],["userN","role1"],...,["userN","roleM"],
 ... ],
```

`acl`

The `acl` properly, when set, shouldbe a JSON array of tuples (triplets), each represented as JSON array of exactly 3 elements. The purpose of this parameter is to define a static access control list, which is merged with the ACL from the database (if any) in the runtime.

For the basic case, each access control entry should be an array of exactly three strings, where first string specifies a role name for which the ACE is intended, second specifies the object name (identifier), and the third specifies the endpoint type. Second and/or third element of an ACE can be `null`, which is interpreted as a wildcard ACE for an object name or endpoint type, respectively (an ACE which matches any object or/and any endpoint type):

```
"acl": [ ...,
  [role :: STRING, object :: STRING | null, endpoint :: STRING | null]
  ... ]
```

For brevity, the notation of Cartesian product is applied to second and third element of the records in `acl` array. That is, like in the previous paragraph,

```
[..., [role, [obj1, obj2,..., objN], [ep1, ep2, ..., epM]], ... ]
  <=> (is equivalent to)
[ ... ,
  [role,obj1,ep1], [role,obj1,ep2], ..., [role,obj1,epM],
  [role,obj2,ep1], [role,obj2,ep2], ..., [role,obj2,epM],
    ...
  [role,objN,ep1], [role,objN,ep2], ..., [role,objN,epM],
 ... ],
```

### 2.7.4 Sample configuration of `authnz` object

An example for configuration of an `authnz` object is given below.

```
{
    "type": "authnz",
    "name": "authnz0",
    "acl_id": 1,
    "realm": "viinex",
    "secret": "c3VwZXJzZWNyZXRcIQ0KRm9vIGJhciBiYXogfiMgNDIK",
    "token_ttl": 36000,
    "cache_ttl": 10,
    "htdigest": "htdigest.txt",
```

```
    "accounts": [
        {
            "type": "password",
            "realm": "viinex",
            "login": "gzh",
            "rem password": "12345",
            "digest": "70ee0684c915cd3d929d3c8ebc9e5162"
        },
        {
            "type": "password",
            "realm": "viinex",
            "login": "root",
            "rem password": "root",
            "digest": "df0d95e30f977056dc1eb706afdb5587"
        },
        {
            "type": "password",
            "realm": "viinex",
            "login": "user",
            "rem password": "viinex",
            "digest": "529fa6e722bff2d02873b1d44908c1b0"
        }
    ],
    "roles": [
        [["gzh","root"],"admin"],
        [["user","user1","user2"],"viewer"],
        ["user3","nobody"]
    ],
    "acl": [
        ["admin",null,null],
        ["viewer",["cam2","cam1"],"SnapshotSource"],
        ["viewer",["cam2","cam1"],["VideoSource","ArchiveVideoSource"]],
        ["viewer",null,["VideoStorage","TimelineProvider",
                        "WebRTC","MetaConfigStorage"]]
    ]
}
```

### 2.7.5 Endpoints implemented by `authnz` object

Endpoint(s) implemented by objects of type `authnz` are:

- 2.9.8 `AuthnzProvider`,
- 2.9.1 `AclClient`

Note that neither of the above endpoint types are publishable. The authentication provider cannot be published via HTTP server or in any other way. When linked with an object which implements the `AuthnzClient` interface, it set up the policy of mandatory authentication and access control.

Also note that the `authnz` object does not modify ACL-related data. If such modification is required, it needs to be performed using the API exposed by respective database object, as

described in section 3.16.

## 2.8 Common configuration sections

For some objects, certain configuration fields share the requirements for their structure. All of such cases above in the description of particular object refer to respective paragraphs of this section.

### 2.8.1 RTP transport priority

For RTP transport negotiation, an object may define the priority for RTP transport protocols. The configuration object member for defining such priority has the following structure:

```
"transport": [TRANSPORT_1, ..., TRANSPORT_N]
```

where `TRANSPORT_k` are strings and may take one of the following values:

- `"udp"` – for RTP over UDP unicast,
- `"tcp"` – for RTP over RTSP over TCP (i.e. interleaved RTP data in a RTSP connection),
- `"mcast"` – for RTP over UDP multicast.

### 2.8.2 Raw video device operation mode

This subsection specifies the syntax of raw video device operation mode description, which is referenced in both configuration of a raw video source in section 2.1.4, and reply to raw video device discovery request in section 3.3.7.

The syntax for raw video device mode description JSON object is given below:

```
{
  "pin": STRING,
  "colorspace": STRING,
  "bpp": INT,
  "planes": INT,
  "framerate": FLOAT,
  "limit_framerate": BOOLEAN,
  "size": [INT,INT],

  "pixel_clock": INT,
  "gain_boost": BOOLEAN,
  "exposure": FLOAT | "auto",
  "flip_horizontal": BOOLEAN,
  "flip_vertical": BOOLEAN,
  "awb": STRING
}
```

The first six properties and the eighth one are mandatory (should be given when the `mode` subsection is authored in the configuration of `rawvideo` object, and are always given in response to raw video device discovery calls).

The `pin` parameter specifies the identifier of the logical "pin" on the device to connect to. The semantics for this depends on the specific device and its driver.

The `colorspace` property specifies the colorspace and format (in-memory pixel layout) of the image buffer. The following values are supported for `colorspace`: `"I420"`, `"YV12"`, `"NV12"`, `"NV21"`, `"YUY2"`, `"YUYV"`, `"UYVY"`, `"YVU9"`, `"RGB"`.

The `colorspace` defines both colorspace (YUV), format (planar/packed), bits per pixel value and pixel data layout in 8 cases of 9, the exception is value `RGB`, when the colorspace is RGB, but the number of planes and the bits per pixel value should be explicitly specified by `planes` and `bpp` parameters. These are only mandatory if `colorspace` equals `RGB`; otherwise their values are ignored and effective number of planes and bits per pixel value are inferred from specified YUV format. For more information on YUV formats see [14].

The `size` parameter should be a 2-element array of integers (a tuple) and specifies the size of image (`[width, height]`).

The `framerate` is a floating-point number specifying the framerate for the video grabber. The semantics of this property differs depending on the context. In the result of rawvideo device discovery calls, the `framerate` contains the value of supprted framerate as reported by the device driver. In the configuration for raw video device, viinex 3.4 attempts to find the mode with a framerate value closest to what is given in the configuration, having all other mandatory parameters matched. That is, this is the only mandatory parameter that may be adjusted by user in configuration with respect to what is output as the result to device discovery call. Note that viinex 3.4 does not pass the adjusted value to the driver, because the latter may consider an adjusted value for framerate as unsupported (depending on driver's implementation). Instead, viinex 3.4 searches for the framerate value, within the list that are explicitly reported by driver as supported, nearest to the `framerate` value specified by user.

This is closely connected with `limit_framerate` parameter, which can only be given in raw video device configuration. By default (when not given), viinex 3.4 interprets as if this value is set to `false`. This means that no additional throttling is performed after the data is acquired from the device driver, and every frame acquired is passed futher to overlay, video encoder, and to linked objects. Then the `limit_framerate` parameter is set to `true`, viinex 3.4 checks whether the timestamps assigned to neighbouring frames acquired from driver differ enough so that the `framerate` specified by the user would not be exceeded if such inter-frame period persists. If this is not the case, i.e. the timestamps of neighbouring frames are too close, – the second frame gets skipped by viinex 3.4. As a result, `limit_framerate` turns on the "software throttling" to limit the effective framerate by the value given in the `framerate` property.

Optionally, image registration parameters can be specified with fields `pixel_clock`, `gain_boost` `exposure`, `flip_horizontal`, `flip_vertical` and `awb`. Pixel clock, if supported, specifies the frequency of data transimission from camera (an integer number, in MHz). Note that depending on the equipment and the driver this value may effectively override the `framerate`. `gain_boost`, if supported, is a boolean value specifying whether the additional gain should be applied to the acquired signal befor the digitizing. The `exposure` is a floating-point value in the range of $[0.0, 1.0]$ which is linearly mapped to allowed range for exposure reported in the runtime by device driver. That is, 0.0 means the smallest possible exposure time, and the value 1.0 means the highest possible exposure time. Setting the `exposure` property to a floating-point value fixes the exposure. Alternatively, the string value `"auto"` can be specified for `exposure` which means the request for auto-exposure functionality of the camera. The

`flip_horizontal` and `flip_vertical` parameters allow for mirroring the image horizontally or vertically. The `awb` parameter can be used to specify the automatic white balance algorithm to be used by camera (if supported). Valid values for this parameter are strings

```
"off"
"greyworld"
"kelvin"
```

The value `"off"` means that AWB algorithm is not applied. The value `"greyworld"` means that the "Grey World" algorithm should be applied for finding the white balance (i.e. the RGB gains are controlled so that the average of color components components have the same value). The value `"kelvin"` means that AWB algorithm constols RGB gains so that resulting image has some predefined color temperature.

If any of optional parameters is omitted from the configuration of `rawvideo` object, the corresponding settings are not performed (left to default or a previous state, depending on the device driver implementation). If any of that parameters is specified but not supported by the device (which is often the case for `pixel_clock`, `gain_boost` and `awb`, because these properties are only available via vendor-specific extenstions to DirectShow), viinex 3.4 may log a warning but proceeds.

### 2.8.3 Video encoder

The `encoder` subsection defines the configuration of H.264 video encoder associated with a raw video source or with a video renderer. The `encoder` subsection has four mandatory fields: `type, quality, profile` and `preset`. The whole `encoder` has therefore the following syntax:

```
"encoder": {
    "type": "cpu" | "qsv" | "cuda" | "vaapi" | "auto",
    "quality": STRING,
    "profile": STRING,
    "preset": STRING,
    "dynamic": BOOLEAN
}
```

The `type` defines the type of encoder implementation. Allowed values for that parameter are:

- `"cpu"`, which stands for software encoder implementation (which uses OpenH264 library),
- `"qsv"`, which stands for Intel-supported QuickSync technology,
- `"cuda"`, which stands for Nvidia-supported CUDA technology,
- `"vaapi"`, which uses Linux VAAPI for hardware encoding acceleration, and
- `"auto"`, which allows viinex 3.4 to choose one of available implementations.

The choice of a hardware-accelerated video encoder in viinex 3.4 is also affected by the environment variable `VNX_HW_ENCODER`. When it's present and is set to "0", viinex 3.4 ignores the presence of any hardware-enabled encoder, and selects the encoder type of `cpu`.

`profile` specifies the H.264 profile for encoder to function in. Allowed values for this property are: `"baseline"`, `"main"` and `"high"`.

The `quality` defines the quantization threshold used by encoding algorithm to throw away the image details while encoding. This affects percieved image quality, resulting stream size, and encoding speed. The acceptable values for `quality` are:

```
"best_quality"
"fine_quality"
"good_quality"
"normal"
"small_size"
"tiny_size"
"best_size"
```

The value `best_quality` means that most details should be preseved while encoding, while the value `best_size` means that most details should be discarded while encoding.

The `preset` parameter specifies the settings of the encoder, in terms of CPU resource consumption. Allowed values for this property are:

```
"ultrafast"
"superfast"
"veryfast"
"faster"
"fast"
"medium"
"slow"
"slower"
"veryslow"
```

These values are listed in order of increase of CPU time required to encode a sequence of frames. On the other hand, the more CPU time is spent on encoding, the better compession rate can be achieved. Note that the `preset` parameter is set "after" the `quality` parameter is fixed. That is, having the quality chosen and fixed, one may choose how much time to spend encoding what should be encoded at chosen `quality`, – and this is done with `preset` parameter. To sum up, `preset` does not affect the resulting image quality, – but the CPU time consumption (`ultrafast` is most lean, `veryslow` is most expensive), and resulting compression rate (`ultrafast` is worst, `veryslow` is best).

There is also an optional boolean parameter `dynamic` in the configuration of an encoder, which indicates whether the encoding of an output video stream should be performed permanently (when `dynamic` set to `false`), or should an encoder only process the stream when at least one client to the encoded video stream exist, – that is, the stream is requested via RTSP or WebRTC or HLS, or it is being written to a video archive. The default value of `dynamic` property is `false`.

Note that video encoder is implicitly used by WebRTC server for sending the video data to the client who requested a stream which is received in H.265 encoding. For compatibility with common browser clients, this is transcoded into H.264 encoding. The video encoder implementation is selected according to the policy `auto`. The presence of `VNX_HW_ENCODER` environment variable is also taken into account.

### 2.8.4 Overlay

The overlay functionality includes the ability to put a static image or HTML page on top of video, and to modify them dynamically over the time via HTTP API described in 3.9. viinex 3.4 uses the `wkhtmltoimage` to render HTML. `wkhtmltoimage` is distributed with viinex 3.4 as a separate binary (viinex 3.4 interacts with that binary via pipes, running `wkhtmltoimage` process whenever an overlay change is requested).

The `overlay` configuration section is common for raw video source and video renderer object. It should be a JSON object,

```
"overlay": {
  "left": INTEGER,
  "top": INTEGER,
  "colorkey": [INTEGER, INTEGER, INTEGER],
  "initial": STRING
},
```

– in which case it defines one overlay, – or an array of JSON objects,

```
"overlay": [
  {
    "left": INTEGER,
    "top": INTEGER,
    "colorkey": [INTEGER, INTEGER, INTEGER],
    "initial": STRING
  },
  ...
],
```

in which case it defines several overlays of which every one can be managed independently from others.

The three mandatory values in JSON object(s) within `overlay` subsection are: `left` and `top` which specify the origin (left-top corner) for the overlay bitmap with respect to the left-top corner of the video, – and the `colorkey`. The latter should be a tuple of 3 integers (represented as JSON array of 3 elements) in range from 0 to 255. These 3 integers specify the color in the overlay bitmap or HTML page to be used as "transparent". Given that color, viinex 3.4 combines the video and the overlay so that pixels on the overlay image or HTML page, having that `colorkey` color value, are not drawn over video, while all other pixels of overlay image are drawn, occluding the image.

The `initial` parameter is optional and specifies the overlay content to be set over the video at viinex 3.4 startup. This property may hold a path to HTML or BMP file. viinex 3.4 automatically recognizes the type of file (based on extension) and performs HTML rendering, if necessary.

In case if `overlay` section is an array, corresponding to multiple overlays, — each of them has its own geometry, color key settings, and the content of a specific overlay can be set independently from other overlays. It is recommended that multiple smaller overlays are used instead of one of a bigger size, for instance, if the goal is to place some text in the opposite corners of the image: the use of several smaller overlays gives better performance.

Note that when rendering HTML, especially if text written in TrueType font, `wkhtmltoimage` uses operating system-dependent rendering algorithms, which include anti-aliasing techniques. This has an effect of mixing the colors of text and the color of background on the edges of the text glyphs. The pixels at the edge of the glyph would have such mixed color which may contain a significant fraction of background tone ("hue"), while being not equal to the background. This is important if the background color equals the `colorkey` value specified in the configuration. viinex 3.4 cannot tell the pixels which were mixed with background color by the renderer from those which were not (this can only be done inside the text renderer, or with an additional "alpha" channel in the image). viinex 3.4 treats such pixels of mixed color as non-background (and therefore non-colorkey, non-transparent). Taking this into account, it is not recommended to use colorkey value that very much differs from text color value. Instead, it's better to choose colorkey close to the text color (not equal, but close). For instance, for text color `#000000` (`[0,0,0]`) it is recommended to use colorkey and background color value `#010101` (`[1,1,1]`): the difference is sufficient for viinex 3.4 to distinguish the background from foreground, while the pixels of "antialiased" colours mixed between `#000000` and `#010101` won't catch an eye. In contrast, if one chooses `#00ff00` value for the HTML background and the colorkey (`[0,255,0]`), the edges of black glyphs will inevitably retain the green colour, which would not be exactly equal to colorkey, but may come arbitrarily close (like `[0,230,0]`) and will be percieved as bright green pixels on the edges of black glyphs. To sum up, it is recommended to choose the "transparent background" color (colorkey) close to (although different from) the font color value.

There is one more hint for rendering text over video, that may turn out useful. To keep the text visible not matter what video background it is rendered over, one may want to have an outline around text glyphs. Such outline can be produced by the similar CSS:

```
<!DOCTYPE html>
<html>
  <meta charset="utf-8">
  <style>
  .outlined
  {
    color: white;
    text-shadow:
    -1px -1px 0 #000,
    1px -1px 0 #000,
    -1px 1px 0 #000,
    1px 1px 0 #000;
   }
  </style>
<body>
  <h1 class="outlined">Outlined text</h1>
</body>
</html>
```

In the above example, the HTML text elements having class `outlined`, have the white color and black outline, which is 1 pixel wide. Such text is clearly visible over virtually any video background, except artificially synthesized images. If this technique is used, the "transparent backgound" color should be chosen close to the color of text *shadow* (which is equal to `#000` in above example, not the inner color `white`), in order to suppress antialiasing-related effect.

### 2.8.5 Video renderer layout

The `layout` subsection of the video renderer configuration, as well as the body of layout change HTTP request described in 3.11.2 should be a JSON object of the form

```
{
  "size": [INTEGER, INTEGER],
  "background": STRING | [INTEGER, INTEGER, INTEGER],
  "nosignal": STRING,
  "viewports": [
    {
      "input": INTEGER,
      "border": [INTEGER, INTEGER, INTEGER],
      "src": [FLOAT, FLOAT, FLOAT, FLOAT],
      "dst": [FLOAT, FLOAT, FLOAT, FLOAT]
    },
    ...
  ]
}
```

The `size` pararmeter is mandatory and defines the width and height of the resulting video, in pixels.

Both width and height should be positive integers. There is one exception to this requirement: width and height may be both set to zero if there's exactly one viewport defined for this layout, and both the source and destination rectangles of the viewport are unit, that is (0,0,1,1). Such configuration would indicate that the layout should produce a picture of the size equal to the size of images coming via the video channel associated with the only specified viewport, without knowing in advance what the actual size is. This is convenient when configuring renderer objects for video analytics or transcoding.

The `background` parameter is optional and can be either an array of three integer values, encoding the RGB colour of the background on the resulting video, or it can be a string representing the path to a JPEG or BMP file on a local filesystem. Note that this is ignored if the layout is changed via the HTTP API: it is not allowed for remote clients to refer to a local filesystem.

If the `background` parameter is not specified in the configuration, the default value of $[0, 0, 0]$ (black background) is assumed. If the `background` parameter is omitted in the HTTP API call, the background previously set is preserved.

The `nosignal` parameter is optional and can be a string representing the path to a JPEG or BMP file on a local filesystem. The image specified with that parameter is displayed in the viewports on the layout where the video source is disconnected (or video data is stalled). If this parameter is not set, the viewports for disconnected/stalled video sources are excluded from the layout. Note that this is ignored if the layout is changed via the HTTP API: it is not allowed for remote clients to refer to a local filesystem.

The `viewports` parameter should be an array of JSON structures, each describing a viewport on the resulting video stream. The viewport is a rectangular zone on the resulting video where one input video stream is rendered. For defining a viewport, the `dst` parameter is required, while `input`, `src` and `border` parameters are optional.

The `input` parameter of the viewport refers to an input video stream which should be rendered in this viewport. This should be an integer zero-based index of the respective video source in

the sorted (lexicographically ascending) list of video sources linked to this instance of video renderer. Such list is known to the client authoring the configuration. This list can also be obtained from an HTTP API call described in 3.11.1.

For instance, if the `links` section of configuration contains the links

```
  "links": { ...
    ["rend0","cam1"],
    ["rend0","cam3"],
    ["rend0","cam2"],
    ...
}
```

between the video renderer and video sources, and there are no more sources linked to that renderer, — then the `"input":0` would refer to `cam1`, `"input":1` would refer to `cam2`, and `"input":2` would refer to `cam3`.

Note that the same `input` value can appear in multiple `viewport`s. It is legal to have viewports more (as well as less) than the number of video sources linked to the renderer. However it also should be understood that while changing the number of viewports is relatively cheap (only one copy/scale operation is added for each viewport), — establishing the link between the video source and the renderer leads to a permanent video decoding, even if current layout contains no viewports with that video source. Also, since the link between the video source and the renderer is established in viinex 3.4 configuration, it is impossible to change the set of such links in the runtime.

If the `input` parameter is omitted from the viewport configuration, this is interpreted as instruction to display an empty viewport containing the `nosignal` image. This can be useful in certain application scenarios when a "placeholder viewport" instead of an actual video should be displayed in the resulting stream.

The mandatory `dst` parameter of the viewport defines the geometry of the viewport on the resulting video stream. The geometry is defined as a JSON array of four floating-point numbers in range from $[0, 1]$, denoting the left, top, right and bottom coordinates of the viewport. The left and right coordinates are measured relatively to the resulting image width, while the top and bottom are measured relatively to the resulting image height.

The optional `src` parameter of the viewport defines the ROI on the source video which should be rendered in the viewport. This ROI is defined as a JSON array of four floating-point numbers in range from $[0, 1]$, denoting the left, top, right and bottom coordinates of the ROI. The left and right coordinates are measured relatively to the respective source image width, while the top and bottom are measured relatively to the respective source image height. Then the `src` parameter is not specified, the default value $[0, 0, 1, 1]$ is assumed, resulting in the ROI of the whole source image.

Given the `src` and `dst` coordinates (in relative units) and the source and destination image size, it it typically the case that the source ROI size in pixels does not match the resulting viewport size in pixels. Even more, it may happen that the aspect ratios of the source ROI and the destination viewport do not match. In either case, what actually viinex 3.4 does is scaling the image of the source ROI so that it fits the viewport on the resulting image, and so that the aspect ratio of the source ROI, in pixels, is preserved when rendering it on the resulting video.

The viewports may overlap on the resulting video. In that case, the visibility of each viewport's content is inferred from the order in which the viewports' description follow in the layout. The

position of a viewport in `viewports` array indicates its "depth" for the purpose of determining the visibility: the viewports at lesser positions within `viewports` array, if overlapped, are occluded by the viewports at greater positions within that array.

The optional `border` parameter of the viewport defines the RGB color components for the border of the viewport. The color is encoded as three integer numbers in range of $[0, 255]$. If the `border` parameter is not given, no border is drawn around the respective viewport.

## 2.9 Endpoint types

Endpoints are fundamental part for understanding how viinex 3.4 configuration is interpreted. Each object mentioned in the `objects` section of the configuration may behave differently and implement different functionality, depending on that object's settings. A formal definition of what behavior should be expected from an object is determined by the set of *endpoints* which respective object is said to be implementing. This set of endpoints is statically inferred from object's configuration, and cannot change in the runtime (or, to be precise, while a viinex 3.4 cluster exists where respective object is instantiated). The set of endpoint types for each object type is given in the above sections, along with the description of config format for each particular object.

This section enumerates the endpoint types and defines semantics (i.e. expected behavior) for an endpoint of every type.

Some of endpoints expose API methods or provide data, and can be published used by external components. Other endpoints only have meaning for other objects within viinex 3.4 cluster. The presence of such endpoints can modify other objects' behavior (when linked against that objects). Within the following sections, such endpoints are marked with the diamond ⋄ sign. Note, however, that while a ⋄ endpoint itself do not expose any public API or provide access to any data, – this does not necessarily mean that the *object* which implements such endpoint does not, as a whole, expose public API or provide access to data. It's typical for objects to implement several endpoints, some of which may provide public API while other may not.

### 2.9.1 `AclClient`

⋄ This endpoint means that the object implementing it could need to access ACL-related data stored in the database.

### 2.9.2 `AclProvider`

This endpoint means that the object implementing it provides read-only access to ACL-related database. Respective API is described in sections 3.16.1–3.16.5.

### 2.9.3 `AclStorage`

This endpoint means that the object implementing it provides means to change the ACL-related data stored within database. Respective API is described in sections 3.16.6 and 3.16.7.

### 2.9.4 ArchiveControl

This endpoint means that the object implementing it provides an API for controlling the contents of a video archive, in particular – it allows for selective video data removal.

### 2.9.5 ArchiveVideoSource

This endpoint means that the object implementing it provides the means for accessing historical data from a video archive.

### 2.9.6 ArchiveVideoSourceClient

◇ This endpoint means that the object implementing it may need an access to historical data from a video archive.

### 2.9.7 AuthnzClient

◇ This endpoint means that the object implementing it may need an access to authentication and authorization provider. This endpoint is implemented by the objects which provide an access to viinex 3.4 instance for users or external software components (HTTP server, RTSP server, WebRTC server).

### 2.9.8 AuthnzProvider

◇ This endpoint means that the obejct implementing it provides methods for authentication and authorization of principals accessing the functionality provided by viinex 3.4 instance.

### 2.9.9 ClusterManagerReader

◇ This endpoint means that the object implementing it may access the functionality exposed by endpoint 2.9.10 ClusterManagerProvider.

### 2.9.10 ClusterManagerProvider

This endpoint provides access to functionality for controlling viinex 3.4 configuration clusters, as described in section 3.19.

### 2.9.11 ConfigReader

◇ This endpoint means that the object implementing it may access the functionality exposed by endpoint 2.9.20 MetaConfigStorage.

### 2.9.12 DynamicVideoClient

◇ This endpoint means that the object implementing it can control video retieval by a video source objects which implement the 2.9.13 DynamicVideoSource. The contract between dynamic video source and its client assumes that a client explicitly subscribes and unsubscribes on live video data from the source, while the dynamic source object begins actual video retreival from its origin if and only if there is at least one dynamic video client.

### 2.9.13 DynamicVideoSource

◇ This endpoint means that the object implementing it can be notified by its clients implementing endpoint 2.9.12 DynamicVideoClient when they subscribe to and unsubscribe from live video data, while the object starts video retreival from origin when there is at least one client, and stops such retrieval when there are no clients left.

### 2.9.14 EventArchive

This endpoint means that the object implementing it can be queried for previously stored events. Respective API is described in section 3.15.

### 2.9.15 EventSink

◇ This endpoint means that the object implementing it is capable of receiveing events from event sources.

### 2.9.16 EventSource

This endpoint means that the object implementing it is capable of emitting events which can be caught by other components or sent to external software, for instance over WebSocket protocol. This endpoint does not expose a callable API though, so to in order for events to be received, this event source needs to be linked with an event sink of an object which could pass the received events to a third party (typically that would be an HTTP server, a script or an external process).

### 2.9.17 KeyValueStore

The object which implements this endpoint is capable of storing arbitrary JSON values organized in a key-value store.

### 2.9.18 KeyValueStoreClient

◇ This endpoint means that an object implementing it needs a 2.9.17 KeyValueStore to function properly.

### 2.9.19 LayoutControl

This endpoint means that the object implementing it exposes the API for controlling the layout for rendering several video sources in a single output video stream. Respective API is described in section 3.11.

### 2.9.20 MetaConfigStorage

This endpoint means that the object implementing it accesses the information on objects within viinex 3.4 cluster, their metaconfiguration, and on endpoints implemented by these objects. This information is published using HTTP API described in sections 3.1.1 and 3.1.2.

Note that for accessing this information a separate privilege (access control entry) matching an endpoint of this type is required.

### 2.9.21 MetricsConsumer

This endpoint is a tag which indicates that the object collects the metrics from other objects — those which expose endpoint 2.9.23 MetricsProducer.

### 2.9.22 MetricsExporter

This endpoint means that the object which exposes it is capable of serializing the metrics collected from other viinex 3.4 object. When published within a 2.9.32 ServicePublisher, the API becomes available on this object to scrape the collected metrics.

### 2.9.23 MetricsProducer

This endpoint means that the object which exposes it is capable of evaluating metrics witnessing about the performance, statistics or other aspects specific to such object.

### 2.9.24 OverlayControl

This endpoint means that the object implementing it exposes the API for controlling the overlay (a graphical layer with usually a transparent background) over the video plane. Respective API is described in section 3.9.

### 2.9.25 PtzControl

This endpoint means that the object implementing it exposes the API for controlling the PTZ device. Respective API is described in section 3.13.

### 2.9.26 RawVideoSource

◇ This endpoint means that the object implementing it represents a source of raw uncompressed video. Note that from perspective of viinex 3.4 infrastructure this endpoint is marked as such which cannot be published. The reason for this is that raw video is published via the local transport mechanism, which is implemented within `vnxvideo` library but is actually an isolated mechanism, its functioning does not involve viinex 3.4 infrastructure.

### 2.9.27 RawVideoSourceClient

◇ This endpoint means that the object implementing can use video stream from a raw uncompressed source. Note that from perspective of viinex 3.4 infrastructure this endpoint is marked as such which cannot be published. The reason for this is that raw video is retrieved via the local transport mechanism, which is implemented within `vnxvideo` library but is actually an isolated mechanism, its functioning does not involve viinex 3.4 infrastructure.

### 2.9.28 RecControl

Objects implementing this endpoint expose the functionality to control the video recording. This functionality is expressed in API described in section 3.6.

### 2.9.29 RecControlClient

◇ This endpoint means that an object implementing it may control video recording.

### 2.9.30 ReplicationSink

This endpoint means that an object implementing it represents the destination point for video replication. In case with managed replication respective object exposes an API described in section 3.7.

### 2.9.31 ReplicationSource

◇ This endpoint means that an object implementing it acts as an active source of rolling (automatic) replication. It does not expose any API; rather it implements a behavior to call methods on a remote viinex 3.4 instance.

### 2.9.32 ServicePublisher

◇ This is an umbrella endpoint meaning that it can be linked with a broad range of other endpoints for the purpose of exposing the functionality implemented by that endpoints. However this endpoint by itself does not expose any API, which means that it cannot provide any means for an external control over how other (linked) endpoints are being published.

### 2.9.33 SnapshotSource

The object implementing this endpoint represents the source of the still images from a video stream, either live or archived. The API which is used to expose the functionality of this endpoint is described in section 3.8.

### 2.9.34 Stateful

This endpoint means that the object implementing it may publish a read-only information which can be retrieved by other components using the API described in section 3.17.1.

### 2.9.35 StreamSwitchControl

This endpoint means that the object implementing it allows for controlling which video stream, of potentially two or more input streams, is being sent as an output, – in other words it allows to select a video stream at server side in the runtime (for instance in video analytics and automation scenarios). Respective API is described in section 3.12.

### 2.9.36 TimelineProvider

The object implementing this endpoint declares that it exposes the functionality to provide other components or external software with information on video archive contents (timeline) for a specific video track. Respective API is described in sections 3.5.2 and 3.7.6.

### 2.9.37 Updateable

This endpoint means that the object implementing it may accept commands to perform some action or update its state, which can be issued by other components using the API described in section 3.17.2.

### 2.9.38 VideoSource

This endpoint means that the object implementing it represents a live media source. It can be further restreamed with WebRTC, RTSP or HLS servers, used by a video analytics engine, stored in a video archive, and so on.

### 2.9.39 VideoSourceClient

◇ The object imeplementing this endpoint declares that it may need an access to a live video source.

### 2.9.40 VideoStorage

The object implementing this endpoint represents a native viinex 3.4 media archive, that is – new media tracks may be created in it in the runtime, and for existing tracks this end-

points acts as the factory for endpoints 2.9.36 `TimelineProvider`, 2.9.33 `SnapshotSource`, 2.9.5 `ArchiveVideoSource` and possibly the 2.9.4 `ArchiveControl`. The APIs for media storage is described in section 3.5.

### 2.9.41 `VideoStorageClient`

◇ The object imeplemting this endpoint declares that it may need an access to native viinex 3.4 video archive (for instance, to enumerate or create archive video tracks in runtime, and so on).

### 2.9.42 `VmsConnection`

◇ The object implementing this endpoint represents a connection (or at least the connection information) to a third party VMS, and may provide with this information the `vmschan` objects linked to that object.

### 2.9.43 `VmsConnectionClient`

◇ This endpoint means that the object implementing it needs the connection to third-party VMS, or at least the information on how such connection should be performed.

### 2.9.44 `WebRTC`

The object implementing this endpoint is a WebRTC server. It exposes an API which is described in section 3.14.

## 2.10 Links

The `links` section contains an array of JSON arrays, each describing one or more functional connections that should be established between viinex 3.4 objects before starting the system (or, to be precise, before starting a viinex 3.4 cluster).

An example for such connection is a connection between a video source and a video archive, established in order to perform the recording of video data from specified video source in the specified video archive.

### 2.10.1 Syntax

The basic form of a link is a pair of names of viinex 3.4 units defined in `objects` section:

```
["object1N", "object2N"]
```

Such pairs of names are encoded as JSON array of strings containing strictly two elements. The order of elements in the pair is not important, that is a link description `[x, y]` is equivalent to `[y, x]`.

There are two more forms for defining a link: the distributive syntax,

```
[["object1N1", ..., "object1Nk"], ["object2N1", ..., "object2Nj"]],
```

and the combinatorial syntax,

```
["objectN1", "objectN2", ..., "objectNk"],
```

The distributive syntax for defining links assumes that instead of a pair of strings, a pair of arrays is specified. Such syntax is interpreted so that all elements from the first array are linked with all elements from the second one. So, specifying one link in distributive syntax like

```
"links": [
 ...
 [["cam1", "cam2", "cam3"], ["rtspsrv0","web0"]],
 ...
]
```

is equivalent to specifying six links of basic form:

```
"links": [
 ...
 ["cam1","rtspsrv0"],
 ["cam2","rtspsrv0"],
 ["cam3","rtspsrv0"],
 ["cam1","web0"],
 ["cam2","web0"],
 ["cam3","web0"],
 ...
]
```

Each of elements in the link of distributive form may be reduced to a single element:

```
"links": [
 ...
 ["stor1", ["cam1", "cam2"]],
 ...
]
```

which is equivalent to just

```
"links": [
 ...
 ["stor1", "cam1"],
 ["stor1", "cam2"],
 ...
]
```

If both arrays are reduced to a single element, this yields in a link of the basic form, i.e. one link between exactly two viinex 3.4 objects.

The combinatorial link syntax allows for defining a link as a “flat” array of more than two elements:

```
"links": [
 ...
 ["rule1", "cam1", "recctl1"],
 ...
]
```

Such syntax is expanded into all possible pairs from specified list, with the restriction that a) an element should not be linked to itself, and b) basic links as pairs are symmetrical. Thus, the example above is expanded to three basic links:

```
"links": [
 ...
 ["rule1", "cam1"],
 ["rule1", "recctl1"],
 ["cam1", "recctl1"],
 ...
]
```

Said that, it's clear that distributive and combinatorial forms of links are only a "syntactic sugar" for specifying multiple basic links. Therefore, for the rest of the document, only the basic links, each between exactly two objects, are discussed. Distributive and combinatorial forms of links are convenient for writing configuration with many congenerous objects but they do not add new semantics to linked objects' interaction, in comparison with what equivalent set of basic links would do.

### 2.10.2 Semantics

While syntactically the links are declared between viinex 3.4 objects, pairwise, it is important to know that actually the links are not established between objects per se. To establish a link means to find matching pieces of functionality in different components, and to make them work together. The elements of functionalify in viinex 3.4 are *endpoints*, their types are described in section 2.9, – and each viinex 3.4 object may implement one or more endpoints. An obvious turn here is that links are established between objects' endpoints. The object's (implementation) type does not matter for establishing a link; it only matters which endpoints does the object implement. Different object types (implementations) may implement endpoints of same type; for the purpose of establishing the link such object types are interchangeable. An example is a video source: `rtsp`, `onvif`, `vmschan` are all different object types, but they all implement endpoint 2.9.38 `VideoSource`, so all objects of that types are interchangeable within `links` section (there is however a qualification to this statement if objects implement more than one endpoint, read on next two paragraphs).

Not every two endpoint types are compatible, and hence not every two objects can be linked together: it depends on objects' endpoint types whether the link between such objects is meaningful in viinex 3.4. For example, there is no point in linking together two video sources, because in no way one video source can make use of another video source (there are no such ways implemented in viinex 3.4). Whether two endpoint types are compatible (in the sense of establishing a link) or not – is known in advance, and the information on that is given in detail in the next section.

Now, a non-obvious part is that when two objects are linked in the `links` section, viinex 3.4 actually attempts to link all endpoints exposed by one object with all endpoints exposed by

other object. All matching endpoint pairs are linked. There is currently no syntax in viinex 3.4 to prevent an endpoint(s) of some specific type(s) from linking. On the other hand, we haven't faced difficulties caused by this limitation yet.

When a link is declared between two objects, it is required that there is at least one pair of endpoint types (one endpoint from each object) that can actually form a link (i.e. at least one pair of endpoints need to be compatible). If a link between two objects is declared but the objects have no compatible endpoints, – this is clearly an error, and such configuration cannot be used (viinex 3.4 would fail to run with such configuration of main cluster, or would refuse to create a secondary cluster with such config). As an example, this would be the case with two video sources: they do not endpoints which would be compatible so that a link could be established. So if one attempts to link two video cameras in viinex 3.4 config – this would be an error.

While some endpoints may participate in many links, the other are "saturable", i.e. may participate in only one link. An example of saturable endpoint is `2.9.7 AuthnzClient`: for instance, a web server cannot use more than one authentication provider. viinex 3.4 detects an attempt to use a saturable endpoint in more than one link, and reports an error in such case. Most endpoints are, however, able to participate in many links.

### 2.10.3 Link types

Despite the syntax of `links` section suggests that there is a "first" and a "second" object in each linked pair of objects, an hence there'll be a first and second endpoint, – that's just an excess of syntax. As the matter of fact a *type of link* between two endpoints is solely determined by the set of types of endpoints being linked, – cardinality of such set is always 2, because there is no endpoint which can be linked with another endpoint of the same type, and currently viinex 3.4 only supports 2-way links. However link type is defined by two endpoint types `P` and `Q` as by a set `{P,Q}`, rather than as by an ordered pair. In other words,

```
forall endpoint types P, Q:
    typeof link (P, Q) == typeof link (Q, P)
        === typeof link {P, Q}.
```

Next paragraphs enumerate some of possible link types and describes their meaning.

## 2.11 Variables interpolation

Throughout all the `objects` section of viinex 3.4 configuration, the configuration properties can be specified not only as a literal JSON expressions, but also as references to variables.

The syntax of a variable reference can have the form of

```
"$(env.NAME)"
```

or

```
"$(var.NAME)"
```

These two options serve for the purposes of reading particular configuration values from environment variables or from external JSON file, respectively.

Whenever the expression of type `"$(env.NAME)"` is encountered as a configuration property value, the environment variable with the name of `NAME` is looked up, and its value is placed as a string JSON value instead of the original expression.

For example, this can be used to avoid storing credentials directly in viinex 3.4 configuration files. A configuration section for a camera could look like this:

```
{
    "type": "onvif",
    "name": "cam4",
    "host": "192.168.32.101",
    "port": 2020,
    "auth": ["administrator","$(env.CAMPASSWD1)"],
    "transport": ["udp","tcp","mcast"],
    "enable": ["video","audio"]
},
```

If there is an environment variable `CAMPASSWD1` set into the password for accessing the camera, – that's the way how that password gets injected into viinex 3.4 config.

The expressions like `"$(var.NAME)"` provide a means to read the parameters from another JSON file, which can be specified at command line (`--variables=`) or in Windows registry. If the variables file is specified, it should contain a JSON dictionary (object). Then, whenever the expression of `"$(var.NAME)"` is encountered in viinex 3.4 configuration property value, it is replaced with a JSON value of the respective key from variables dictionary. Note that in case of `"$(var.NAME)"` interpolation, substituted values should not necessarily be strings, – they can be JSON values of any type.

This feature is considered as a way to provide some local, site-dependent or environment-dependent parameters to viinex 3.4 configuration, – so there is no API to set the "variables" dictionary, and the same is true for environment variables.

## 2.12 License information

viinex 3.4 requires a license document for functioning. A license document could be local, represented by a string of characters in viinex 3.4 configuration, Windows registry or a file on Linux filesystem; a license document can also be stored on a USB device (SenseLock dongle). Another option for a viinex 3.4 instance is to not have neither a local license document, nor an access to the Senselock dongle, but to have an access to another (remote) viinex 3.4 instance running a floating license server. Finally, there is an option to use provisional licenses, where viinex 3.4 automatically connects to Viinex key management portal, requests a license key and installs it.

Which of above options is chosen depends on the presence of `license` key in viinex 3.4 configuration, on the contents of this key, and on the visibility of a Senselock USB device and the presence of "emulated" license storage. The algorithm to choose between these options is as follows:

1. If the `license` key is present in viinex 3.4 configuration, the content of this key is used. Depending on format of the value for the `license` key, this can be either a local

"software" license (i.e. a license bound to the local harware), or a reference to a floating license server.

2. If the `license` key is absent in viinex 3.4 configuration and a Senselock USB dongle is accessible by the viinex 3.4 instance, – then the license document contained in that dongle is used.

3. If the `license` key is absent in viinex 3.4 configuration and no Senselock USB dongle is accessible by the viinex 3.4 instance, – then the "emulated" license storage is checked for the presence of a license document.

The control over a license documents that reside in the Senselock USB dongle or a "emulated" license storage is performed using `viinex-lm-upgrade` utility and described in section 7.2. Either of these cases specify a local license document, but do not interfere with viinex 3.4 configuration (i.e. they are controlled independently from viinex 3.4 configuration). This section describes the first of mentioned three options, when the `license` property is present in viinex 3.4 configuration.

### 2.12.1 Local license document

The `license` key of configuration document is optional. However, if present, it overrides all license information that might be stored in USB dongles attached to the server or in the "emulated" license storage. When the license `license` key is absent, license information is taken from USB dongle or "emulated" license storage.

The value stored under `license` key of the configuration document may be a "license document" – a string containing the encrypted license information together with information about hardware where an instance of viinex 3.4 is allowed to run on. License documents are generated by viinex 3.4 licensor upon request. This string value is actually of the same kind which is accepted as input to command `viinex-lm-upgrade update`, as described in section 7.2. The difference that encrypted license documents intended for use with USB dongle, are generated to be bound to specified USB dongle, and cannot be placed in `license` section of configuration document. And vice versa, the encrypted license documents generated to be placed in `license` section of the configuration, are always bound to the PC hardware, they are generated by licensor in response to information on PC hardware where viinex 3.4 is supposed to run, and they cannot be used to update a USB dongle.

The example of viinex 3.4 configuration with a local license specified in it could be as follows:

```
{
    "objects": [ ... ],
    "links": [ ... ],
    "license": "NZOiBGeL...(quite a long base64 string)..8zjmkfw=="
}
```

### 2.12.2 Floating license client

To access a floating license server, the `license` property of viinex 3.4 configuration may hold a JSON object containing the information to access the HTTP endpoint where respective license server is exposed:

```
{
    "objects": [ ... ],
    "links": [ ... ],
    "license": {
        "url": STRING,
        "auth": [LOGIN, PASSWORD],
        "certificate": FILEPATH
    }
}
```

The only mandatory parameter here is `url`. It should contain the HTTP or HTTPS URL of the floating license server object. For a sample configuration given in section 2.5.8, where the floating license server has the name `licmgr0` in viinex 3.4 configuration, the URL would look like

```
http://SERVER:PORT/v1/svc/licmgr0
```

where `SERVER` and `PORT` are respectively the host name or IP address where floating license server runs, and the port number of viinex 3.4 web server. If a web server uses TLS, the "http://" schema should be replaced with "https://".

An optional `auth` parameter, if present, should contain a login name and password for authentication in viinex 3.4 web server.

An optional `certificate` parameter, if present, should contain a path to a file on a local filesystem, represeting a TLS certificate to check server's authenticity first, before authenticating ourselves against that server. This is recommended, especially for setups where the floating license server is accessible in Internet. If the `certificate` parameter is omitted and TLS is still being used, then the server's authenticity is not being checked by floating license client.

For simple setups, when the web server where floating license server runs does not require authentication, only the `url` parameter is required, and its value may be placed directly as the value of `license` parameter of viinex 3.4 configuration, similiar to the following:

```
{
    "objects": [ ... ],
    "links": [ ... ],
    "license": "http://192.168.0.71:8880/v1/svc/licmgr0"
}
```

In such case viinex 3.4 distinguishes between a local license document and an URL pointing to a floating license server by the presence of `http://` or `https://` prefix in the `license` string value.

### 2.12.3 Provisional licenses

The mechanism of provisional licenses assumes that the user has an active account in Viinex license key management portal (), has created an API key for accessing the portal, and has a positive balance on their account. Here, positive balance means that licenses were credited onto user's account, so it is possible for that user to generate a permanent license key for some specific server. The mechanism of provisional licenses suggests that this credited amount of licenses is

kept unspent – then respective number of provisional licenses can be generated. Provisional license key is temporary, it expires in 30 days, and upon expiration, viinex 3.4 automatically requests the new provisionial key. This happens seamlessly, without any downtime. The major advantage of such approach for the user is that viinex 3.4 license does not need to be permanently bound to a specific server, as this would be the case with perpetual license key. This allows viinex 3.4 software to be migrated from one server to another if this is required.

An instance of viinex 3.4 which uses provisional licensing may serve as a floating license server for other viinex 3.4 instances in the local network. The concepts of provisional licenses and floating licenses are similar, although there are differences:

- Floating licenses are completely transient; floating license client does not store any state; all licenses acquired at startup are returned when floating license client shuts down, and on re-start floating licenses are re-acquired from server. (These licenses are transferred between floating license server and client in form of ephemeral documents, internally called "tickets", rather than real viinex 3.4 license keys.
- Respectively, floating license server tracks how many clients requested floating licenses, but only does this in RAM. There is no persistent state containing this information. As a downside, there should not be connectivity loss between floating license client and server for longer than 8 hours.
- In contrast, when a viinex 3.4 instance uses a provisional licensing mode, it requests a real license key from key management portal. This license key gets installed into the local license store. Upon next startup, viinex 3.4 does not have to contact license key management portal, – rather it would use an installed provisional license key. This would be the case until provisional license expires (recommended and maximum expiration time for provisional licenses is 30 days).
- Respectively, Viinex license keys management portal is not as permissive as floating license server: it cannot be restarted (to effectively nullify the information on previously acquired licenses); the balance of each account is persistent, and when provisional license is requested and granted, – this information is stored at the portal, so that the user cannot "double spend" credited licenses. This means that before a request for provisional license key gets processed, the portal checks whether the client has sufficient credit, not (yet) spent on perpetual licenses.

To use the mechanism of provisional licenses, one needs to specify the following in the `license` section of viinex 3.4 configuration:

```
"license": {
  "api_key_id": NUMBER,
  "api_key": STRING,
  "features": [ [STRING, NUMBER] ],
  "days": NUMBER
}
```

Here `api_key_id` and `api_key` are credentials retrieved within Viinex license key management portal, `features` should contain the list of licenses to be requested, along with the quantities, and `days` should be the number of days for which provisional license is requested. This should not exceed 30, however a user may want to reduce it in certain usage scenarios. (Essentially this allows new provisional license to be requested sooner).

The value of `features` configuration property should have the same form as printed out by `viinex-lm-upgrade show` command with the command line switch `-p0`, for example:

```
    "features": [
      [["ViinexCore", 1]
      ,["IpVideochannel", 24]
      ,["VideoArchive", 4]
      ,["VideoRenderer", 24]
      ,["ReplicationSink_Channel", 8]
      ,["ReplicationSource", 4]
      ,["ReplicationSink", 4]
      ,["VmsChan", 24]
      ,["UsbVideochannel", 4]
    ]
```

The licenses for specified features are requested from license key management portal all at once, no matter how many objects are specified in the configurtation and if these licenses are currently needed. The logic behind this is that a) licenses may be required later, as new clusters are created, and b) this instance of viinex 3.4 may also host floating license server for a local network, in which case it's also impossible to compute how many licenses of which types are needed. It's user's responsibility to provide license types and quantity sufficient for further operation of the viinex 3.4 instance which uses provisional licensing.

Provisional licenses do not affect user's ability to issue a perpetual license key on the Viinex license key management portal. However, once the licenses are "spent" for issuing a perpetual license, – the will no longer be available for issuing new provisional license keys. To continue with provisional licenses, the user may want to top up their account's balance in the portal.

If you would like to try out how the provisional license mechanism works, please reach out to viinex support team for a test API key to access the keys.viinex.com portal.

An example for provisional license section in viinex 3.4 configuration is provided below:

```
{
    "license": {
        "api_key_id": 1000011,
        "api_key": "...redacted...",
        "features": [
            ["ViinexCore",11],
            ["IpVideochannel",2],
            ["VideoArchive",44],
            ["VideoRenderer",2],
            ["ReplicationSink",44],
            ["LicenseManager",1]
        ],
        "days": 7
    }
}
```

## 2.13 Split configuration

For convenience of deployment and for better manageability, the configuration document for viinex 3.4 can be split into several separate files. For this, it is required that the files are put into one directory, and have `.json` extension. If the path to a directory is passed to viinex 3.4 at startup, the latter automatically reads all `*.json` files in that directory, merges their content according to rules described below, and interprets the merged result as a configuration document to start up with.

It's up to the user to decide which parts of configuration to put into each separate file. All files should have the same configuration document format that is described in this chapter, that is – each file should be a JSON document that can contain any of three optional keys – `objects`, `links` and `license`. The rule for merging `objects` and `links` sections of configuration parts is to concatenate corresponding JSON arrays. The order of elements in that arrays does not matter, therefore the resulting merged configuration will contain all `objects` and all `links` mentioned in configuration parts.

For example, two configuration parts, — file `part1.json`:

```
{
    "objects":
    [
        {
            "type": "TYPE1",
            "name": "NAME11",
            "parameter1": "value1"
        },
        {
            "type": "TYPE2",
            "name": "NAME12",
            "parameter2": "value2"
        }
    ],
    "links":
    [
        ["NAME11", "NAME12"]
    ]
}
```

and file `part2.json`:

```
{
    "objects":
    [
        {
            "type": "TYPE1",
            "name": "NAME21",
            "parameter1": "value3"
        },
        {
            "type": "TYPE2",
            "name": "NAME22",
```

```
            "parameter2": "value4"
        }
    ],
    "links":
    [
        ["NAME21", "NAME22"]
    ]
}
```

would be merged into an equivalent of the following configuration:

```
{
    "objects":
    [
        {
            "type": "TYPE1",
            "name": "NAME11",
            "parameter1": "value1"
        },
        {
            "type": "TYPE2",
            "name": "NAME12",
            "parameter2": "value2"
        },
        {
            "type": "TYPE1",
            "name": "NAME21",
            "parameter1": "value3"
        },
        {
            "type": "TYPE2",
            "name": "NAME22",
            "parameter2": "value4"
        }
    ],
    "links":
    [
        ["NAME11", "NAME12"],
        ["NAME21", "NAME22"]
    ]
}
```

The rule for merging the `license` section is to find an existing `license` value, if any, within the partial configuration files, and to use it in the resulting merged configuration document. That is, if no `license` values were present in configuration parts, it will not be present in the merged configuration. If exactly one `license` value was present in exactly one configuration part, – that value will be present in the resulting merged configuration. If more than one `license` section is present in configuration parts, it is guaranteed that one of them will be present in the resulting configuration document, but it is undefined which one, therefore it's advised to avoid authoring configuration parts in such way. The recommended practice is to put the PC hardware-bound license document into one separate partial configuration file, containing `license` section only.

Splitting the configuration document into several files can be used to isolate independent parts of configuration, or to separate rarely changing parts from those which change often.

An example layout for configuration document split into several files is given below.

File `web.json`:

```
{   "objects": [
        {   "type": "webserver",
            "name": "web0",
            "port": 8880,
            "staticpath": "static"
        }
    ]
}
```

The web server is defined within this configuration part. It's likely that, once created, this part would never be changed.

File `storage.json`:

```
{
    "objects": [
        {
            "type": "storage",
            "name": "stor0",
            "folder": "C:/videostorage",
            "filesize": 16,
            "limits": {
                "keep_free_percents": 20
            }
        },
        {
            "type": "recctl",
            "name": "recctl1",
            "prerecord": 5,
            "postrecord": 3
        },
        {
            "type": "recctl",
            "name": "recctl2",
            "prerecord": 5,
            "postrecord": 3
        }
    ],
    "links": [
        ["stor0","web0"],
        ["recctl1","stor0"],
        ["recctl2","stor0"],
        ["recctl1","web0"],
        ["recctl2","web0"]
    ]
}
```

A video archive and two recording controllers are defined in this configuration part. The recording controllers are attached to the video archive, and they all are exposed via the web server. Note that video sources are not mentioned in this part of configuration yet.

File `zone1cams.json`:

```
{
    "objects": [
        {
            "type": "rtsp",
            "name": "cam1",
            "url": "rtsp://192.168.0.121:554/ISAPI/streaming/channels/101",
            "auth": ["admin","12345"],
            "transport": ["udp"]
        },
        {
            "type": "rtsp",
            "name": "cam2",
            "url": "rtsp://192.168.0.111:554/ISAPI/streaming/channels/101",
            "auth": ["admin","12345"],
            "transport":["tcp"]
        }
    ],
    "links": [
        ["cam1", "recctl1"],
        ["cam2", "recctl1"],
        ["cam1", "web0"],
        ["cam2", "web0"]
    ]
}
```

With this file, two video cameras are created, their video streams are published via the web server, and the newly created cameras are recorded with `recctl1` object controlling the recording process. Note that this part of configuration does not mention a video archive directly. This is how the configuration can be organized into a number of small sub-configurations of "scenes".

File `license.json`:

```
{"license":"QIXOGBdR1qv.....BSXOKX2DHY="}
```

As recommended above, the license key is written into a separate file to ensure that there is only one license document stored in the configuration, so it's guaranteed to be replaced upon license upgrade. On the other hand, license upgrade would only touch one file, preserving the rest of configuration unchanged.

# 3 HTTP API

Functionality provided by viinex 3.4 is available for use from client applications by means of HTTP API. When a component of viinex 3.4 is published in the built-in viinex 3.4 web server, it exposes its specific part of API under that web server. Note that a component may be configured to not expose its HTTP API; in that case it only participates in internal interaction with other viinex 3.4 components.

All components' API are published under URLs depending on the names of components in viinex 3.4 configuration document. For instance, when a recording controller with name "recctl1" is published in the web server, its API becomes available under subtree

```
http://SERVERNAME:SERVERPORT/v1/svc/recctl1
```

This is a general rule: the common prefix of `/v1/svc/`*`NAME`* is prepended to the URLs for accessing functionality of viinex 3.4 component with name `NAME`. One exception from this rule is the web server itself, which is uniquely identified by TCP port number.

The HTTP APIs for all previously mentioned viinex 3.4 components is described in following sections. In the tables following, in rows "Request URL and applicable methods", the parts of URLs that are specific to described service/function, are highlighted with *italic font*.

## 3.1 Web server

### 3.1.1 Enumerate published components

**Request purpose**

Obtain the list of components published under the web server, along with their endpoint types.

**Request URL and applicable methods**

`GET http://servername:`*`port`*`/v1/svc`

**Request parameters**

none

**Response syntax**

JSON array of 2-element arrays (tuples):

```
[ ["endpointType1", "componentName1"],
  ...,
  ["endpointTypeN", "componentNameN"] ]
```

#### Response example

```
[
  ["VideoStorage", "stor0"],
  ["VideoSource", "cam1"],
  ["VideoSource", "cam2"],
  ["SnapshotSource", "cam2"]
]
```

means that there are four components published under the web server: video archive, and two video sources. Note that while viinex 3.4 configuration file contains definition for object types, these are essentially implementation types. One object may, however, implement several endpoints, which can be exposed (or not exposed) in HTTP API. In this request, endpoint (interface) types are described, not object (implementation) types. For more information on endpoint types see section 2.9.

#### Remarks

This API call is exposed as a part of endpoint `2.9.20 MetaConfigStorage`. If the HTTP server on which this call is made is linked with an authentication and authorization provider, then the separate privilege is required to perform this call. Moreover, what's important, the results of this call are filtered for each principal according to the principal's permissions. Of all the objects and endpoints created within this viinex 3.4 cluster, only those are included with response to this call, which are available to the principal (caller).

### 3.1.2 Obtain the metainformation on published components

#### Request purpose

Obtain the metainformation previously stored in the configuration sections for objects created and published under this instance of web server.

#### Request URL and applicable methods

`GET http://servername:port/v1/svc/meta`

#### Request parameters

none

#### Response syntax

JSON object (associative array) with keys equal to viinex 3.4 object `name`s, and values equal to the content of `meta` property in configuration of respective viinex 3.4 object:

```
{
  "componentName1": JSON_VALUE_1,
  ...
  "componentNameN": JSON_VALUE_N
}
```

Only the components are reported which are published under this instance of viinex 3.4 web server, and which have their `meta` property set to a non-null JSON value.

#### Response example

For the configuration example given at page 13, the response to metainformation request would look like:

```
{
  "cam1": { "desc": "Backyard" },
  "cam2": { "desc": "Hall" },
  "stor0": { "desc": "Long-term storage",
             "volume": "/dev/sdb" }
}
```

It's up to the user or the application which authors viinex 3.4 configuration to decide how to use these values. viinex 3.4 only reads them from configuration documents and reports them unmodified, in response to this `/v1/svc/meta` request.

## 3.2 Authentication

### 3.2.1 Authentication challenge

#### Request purpose

Receive an authentication challenge from the server

#### Request URL and applicable methods

`POST http://servername:port/v1/`*`authGetChallenge`*

#### Request parameters

login – user login in case of `password` authentication or API key in case of `apikey` authentication record type to be used on the next step

#### Response syntax

JSON object:

```
{
    "when": TIMESTAMP,
    "realm": STRING,
    "challenge": STRING,
    "signature": STRING
}
```

where `when` – timestamp indicating when the challenge was issued, `challenge` – the challenge itself (that's basically a random data), `signature` – the signature of the challenge, which allow the server to validate client's response to the challenge. `signature` is hashed and signed mix of timestamp when challenge was issued, the challenge data and the login of a client who requested the challenge. This makes for a server possible to verify that client has responded to the challenge that was proposed to that client, – not to some other challenge that might have been tampered or stored previously and outdated by the time of client's response.

The `realm` string is optional and returned only in case if the `login` parameter referenced the account with `password` authentication data record type, see section 2.5.5 for details. `realm` is returned for client's convenience, it can be used by client to perform the computations with user's password to build the correct authentication response. If authentication record type referenced by `login` parameter is `apikey`, then the secret for this account is stored by server and client as plain text, therefore no additional hashing of password is required before computing the authentication response by client. In such case, `realm` parameter is absent in server's challenge.

#### Response example

Request:

```
POST http://localhost:8881/v1/authGetChallenge?login=agentA
```

Response:

```
{
    "when":"2017-01-16T12:06:46Z",
    "challenge":"37016c28c4dceb8b813b9fc246c66200addc439398f428\
                0a1a49db3378b03dda",
    "signature":"b541cafadd67e364c653c29d6b49c0dc"
}
```

`signature` is hash-based message authentication code [7] of a combination of `login`, challenge timestamp and challenge data.

### 3.2.2 Authentication response

#### Request purpose

Authenticate client at the server

### Request URL and applicable methods

`POST http://servername:port/v1/`*`authCheckResponse`*

### Request parameters

Authentication response should be passed as POST request body.

### Response syntax

Authentication response which should be passed as an argument to this call as the HTTP request body is JSON document with fields `login, when, challenge, signature` and `response`. The first four fields have the same meaning as in challenge description, and their values should be preserved by client after it receives an authentication challenge from server.

The `response` field should be the HMAC [7] of `challenge` data, computed with client's `secret`, if the authentication data record type is `apikey`, or hashed combination of user's password and realm, if authentication data record type is `password` (see section 2.7.3 for details). In other words, `secret` is in either case the thing stored by server, and `response` is computed as

```
hex(HMAC(challenge.challenge, secret))
```

There is a reference implementation for this algorithm available at [24].

Upon successful authentication, server sets the cookie `auth` which contains the information on authenticated client's identity and permissions (i.e. an access token). This cookie has the JWT (JSON Web Token) format, which is described in [25].

Upon authentication failure, server returns HTTP error code 401.

The client should compute its `response` to a server's authentication challenge as HMAC-MD5 [7] of the `challenge` data. The secret for computing the HMAC is `secret` or MD5 hash of user's password combined with the `realm` (in such case, the realm is passed in server's response to `authGetChallenge` request). After computing HMAC, it should be converted to base16 (a string of hexadecimal digits).

For instance, with `secret` equal to `foobarsecret42` and `challenge` equal to

```
fdfeefeaa33d4f683bc843cae4375592439fa980ac969e7757226baf15ef5398,
```

client's response should be equal to `4c7bd5ae85894f78eb87bd2955f4cd83`.

The cookie set by the server in case of successful authentication, should be passed by the client with each API request. This happens automatically when using a web browser as HTTP client, but may be required to be done explicitly when building a programmatic HTTP client.

The server may treat the authentication token as temporary on its discretion, and require the client to repeat the authentication at any time (returning HTTP error code 401).

Authentication is optional for using viinex 3.4 public API. To turn authentication off, one may remove the link between an HTTP server object and the authentication and authorization

provider object in the `links` section of the configuration, as described in sections 2.5.5 and 2.7.

## 3.3 Environment

This subsection contains description for API calls for finding out the details of environment where viinex 3.4 server is running in, in particular — accessible hardware, like attached disk drives, visible ONVIF cameras, attached USB dongles, etc.

### 3.3.1 Attached SenseLock USB dongles

#### Request purpose

Obtain the list of currently attached SenseLock USB dongles' serial numbers.

#### Request URL and applicable methods

`GET http://servername:port/v1/env/`*`senselock`*

#### Request parameters

none

#### Response syntax

JSON array of strings, each representing the identifier of a dongle attached to the computer:

```
[ "senslockId1", ..., "senslockIdN" ]
```

#### Response example

```
["9529520000003638"]
```

denotes that there's one SenseLock USB dongle with serial number 9529520000003638 currently attached to the server. The resulting array is empty if no USB dongles found.

### 3.3.2 License document content

#### Request purpose

Obtain information on current status of Viinex license manager, that is the license document and the mode which license manager is started in.

### Request URL and applicable methods

`GET http://servername:port/v1/env/`*`license`*

### Request parameters

none

### Response syntax

JSON object of the following structure:

```
{
  "document": {
    "product": "Viinex20",
    "binding": {"senselock": STRING} | {"hwid": STRING},
    "features": { STRING_1: INT_1, ..., STRING_N: INT_N },
    "timelimit": TIMESTAMP
  },
  "mode": "hardware" | "software" | "demo"
}
```

The `document` section denotes the content of license document with which license manager is started. It is a JSON object with four fields:

`product` is a constant string equal to "Viinex20" for viinex 3.4. `binding` is a JSON object which denotes either senselock dongle identifier, or a hardware ID string describing the computer harware which license document was issued for (see also sections 7.2.1, 7.2.3). The `features` field is an associative array with keys of type string, each corresponding to a position in the license document, and values of type integer, denoting the limit for the respective position. The positions include:

- `IpVideochannel` – an object which implements video registration functionality via TCP/IP network (RTSP protocol), that is – RTSP video source or an ONVIF video camera;
- `UsbVideochannel` – an object which implements video registration functionality via DirectShow or Video4Linux API;
- `VmsChan` – the `vmschan` object providing an access from viinex 3.4 video channel in a 3rd party VMS;
- `ReplicationSink` – video archive replication sink (i.e. the server capable of gathering video information from multiple viinex 3.4 video archives;
- `ReplicationSource` – video archive replication source, that is – an agent capable of sending video information from a local archive to the replication sink.

The `timelimit` field, if set, denotes when license document becomes invalid. If `timelimit` field is not set or equals to `null`, this means that the license document is permanent.

The `mode` field describes the current mode of license manager operation. There are three possible values for that field:

- `hardware` – license manager runs on the USB dongle, bound to that dongle;
- `software` – license manager runs on the PC, bound to the hardware parts of that PC;
- `demo` – license manager runs with predefined limitations, not bound to this specific PC or a USB dongle.

**Response example**

```
{
  "document": {
    "product": "Viinex20",
    "binding": {
      "senselock": "9529520000003638"
    },
    "features": {
      "IpVideochannel": 16,
      "ReplicationSink": 1,
      "ReplicationSource": 2
    },
    "timelimit": null
  },
  "mode": "hardware"
}
```

means the license manager operates on the USB dongle. License document is bound to the USB dongle with identifier `9529520000003638`; it is permanent, and allows for creation of 16 RTSP video sources or ONVIF cameras, 1 replication sink, and 2 video replication sources.

### 3.3.3 Probe for licenses

**Request purpose**

Ask the license manager whether the next attempt to acquire the licenses for specified features in specified quantity would succeed.

**Request URL and applicable methods**

`POST http://servername:port/v1/env/`*`license/probe`*

**Request parameters**

Request body – a JSON array describing feature names and respective quantities to probe

**Response syntax**

The request body should contain a JSON array of pairs of feature name and respective quantity, each pair encoded as follows:

`[ [[STRING1, null], INT1], [[STRING2, null], INT2], ... ].`

This format has been changed as of viinex version 3.4 in order to enable the reuse of licenses by objects referring to the same physical IP camera or VMS channel. The feature names accepted in this request are the same values that are described in section 3.3.2.

The call returns an empty JSON object `[]` upon success. Upon failure, an object containing boolean property `"success"` set to `false` and the string property `"error"` is returned.

**Response example**

```
$ curl -X POST http://localhost:8880/v1/env/license/probe \
    --data-binary '[["IpVideochannel",100]]'
[]

$ curl -X POST http://localhost:8880/v1/env/license/probe \
    --data-binary '[["VmsChan",1000]]'
[]

$ curl -X POST http://localhost:8880/v1/env/license/probe \
    --data-binary '[["IpVideochannel",10000]]'
{"error":"Insufficient licenses for IpVideochannel","success":false}

$ curl -X POST http://localhost:8880/v1/env/license/probe \
    --data-binary '[["IpVideochannel",1000],["VmsChan",100500]]'
{"error":"Insufficient licenses for VmsChan","success":false}
```

In the first two examples the license manager is probed for 100 and 1000 licenses for the `IpVideochannel` feature, and both requests succeed. In the 3rd and 4th examples viinex 3.4 instance is probed for a large number of licenses, and both requests result in a failure, describing the reason in each case.

**Remarks**

This call may be used to rougly estimate the number of specific licenses left (available) to use. The call itself does not affect the state of viinex 3.4 license manager, i.e. the licenses that are probed for are not actually acquired by this call. On the other hand, in a multi-user and multi-tasking environment, when there can be many tasks concurrently acquiring and releasing licenses (like in the scenarios with dynamic IP video channels, or when the configuration clusters may be concurrently created or destroyed), – the result of this call should be treated as possibly outdated right after it was produced, and therefore it should not be used in any precise computational logic. A recommended way of using this API is to issue some number of `license/probe` calls in order to obtain a very rough estimate of how many licenses for a specific feature are left (in terms like – zero, less than 5, less than 10, ..., more than 1000), and present this conclusion to the user, so that he could take a decision on creating or freeing certain instances of viinex 3.4 objects.

It should also be taken into account that depending on license manager implementation and its current mode of functioning, this call may result in an interaction with a remote system, and therefore may cause some arbitrary timeouts.

### 3.3.4 Obtain viinex 3.4 software version

#### Request purpose

Obtain the information on viinex 3.4 version and build number.

#### Request URL and applicable methods

`GET http://servername:port/v1/env/about`

#### Request parameters

none

#### Response syntax

In response, the JSON object with string properties `product`, `version`, `build` and `version_full` is returned:

```
{
  "product": STRING,
  "version": STRING,
  "build": STRING,
  "version_full": STRING
}
```

#### Response example

Example call to `/v1/env/about` API method could yield the result similiar to the following:

```
{
  "product":"viinex",
  "version":"2.0.0",
  "build":"171",
  "version_full":"2.0.0.171"
}
```

### 3.3.5 Discover visible ONVIF devices

#### Request purpose

Obtain the list of ONVIF devices visible by the server being asked.

### Request URL and applicable methods

`GET http://servername:port/v1/env/`*`onvif`*

### Request parameters

none

### Response syntax

An array of JSON objects, each having two fields `scopes` and `xaddrs`:

```
[
  {
    "scopes": {
      "key1": "val1",
      ...
      "keyN": "valN",
    },
    "xaddrs": [
      "URI_1",
      ...
      "URI_M"
    ]
  },
  ...
]
```

The `scopes` member is an associative array containing some of scopes put by ONVIF device into the WS-Discovery response, that are recognized by viinex 3.4, see remarks below for more details. The `xaddrs` member is an array of strings representing the URIs for accessing ONVIF device, as reported by WS-Discovery response. Appropriate elements of this array can be used as parameter for `onvif/probe` call, and for ONVIF video source configuration in viinex 3.4.

### Response example

```
[
  {
    "scopes": {
      "location": "city/hangzhou",
      "name": "HIKVISION%20DS-2CD2132-I",
      "hardware": "DS-2CD2132-I"
    },
    "xaddrs": [
      "http://192.168.0.111/onvif/device_service",
      "http://[fe80::4619:b6ff:fe6a:4380]/onvif/device_service"
    ]
  },
  {
```

```
    "scopes": {
      "location": "city/hangzhou",
      "name": "HIKVISION%20DS-2CD4024F",
      "hardware": "DS-2CD4024F"
    },
    "xaddrs": [
      "http://192.168.0.121/onvif/device_service",
      "http://[fe80::4619:b6ff:fe6b:2b45]/onvif/device_service"
    ]
  }
]
```


In the above example, two ONVIF devices were discovered, having 192.168.0.111 and 192.168.0.121 IPv4 addresses.

#### Remarks

Section 7.3.2.2 of ONVIF core specification [8] requires that "scopes" are represented by URIs in form of `onvif://www.onvif.org/<path>`. For convenience, viinex 3.4 chops off the permanent part of that value, leaving the significant part only. Resulting `scopes` associative array may be used to display human-readable values for discovery results in the UI.

viinex 3.4 recognizes the `name`, `hardware` and `location` scope values filled in by ONVIF device in WS-Discovery response.

### 3.3.6 Probe an ONVIF device

#### Request purpose

Obtain the detailed information on video sources and profiles configured on the ONVIF device.

#### Request URL and applicable methods

`POST http://servername:port/v1/env/`*`onvif/probe`*

#### Request parameters

There are no parameters in the request URL. A request should carry the body containing a JSON object with one of two mandatory fields, `url` or `host` (in the latter case it can be accompanied by optional field `port`), and an optional field `auth` which may contain string array of length two: the login name and the password for accessing the ONVIF device.

#### Response syntax

Request body:

```
{ "url": "STRING", "auth": [STRING, STRING] }
```

or

```
{ "host": STRING, "port": INT, "auth": [STRING, STRING] }
```

Response body:

```
{ "info": OBJECT, "video_sources": OBJECT, "profiles": OBJECT }
```

See remarks below for more details.

### Response example


```
{
  "info": {
    "vendor": "HIKVISION",
    "serial": "DS-2CD2132-I20140823CCWR477543489",
    "model": "DS-2CD2132-I",
    "firmware": "V5.2.0 build 140721"
  },
  "video_sources": {
    "VideoSource_1": {
      "token": "VideoSource_1",
      "framerate": 25,
      "resolution": [ 2048, 1536 ]
    }
  },
  "profiles": {
    "Profile_1": {
      "token": "Profile_1",
      "name": "mainStream",
      "fixed": true,
      "video": {
        "codec": "H264",
        "resolution": [ 1280, 720 ],
        "source": "VideoSource_1",
        "quality": 4,
        "bounds": [ 0, 0, 2048, 1536 ]
      }
    },
    "Profile_2": {
      "token": "Profile_2",
      "name": "subStream",
      "fixed": true,
      "video": {
        "codec": "H264",
        "resolution": [ 704, 576 ],
        "source": "VideoSource_1",
        "quality": 3,
        "bounds": [ 0, 0, 2048, 1536 ]
      }
    }
```

```
  }
}
```

**Remarks**

A response to `onvif/probe` call contains three members: `info`, `video_sources` and `profiles`.

The `info` member is a JSON object of the form

```
{
    "vendor": "STRING",
    "model": "STRING",
    "serial": "STRING",
    "firmware": "STRING"
}
```

and contains general information on the ONVIF device being probed: its vendor, model name, device serial number and firmware version.

The `video_sources` member is an associative array with keys equal to video sources' "tokens" (identifiers within a single device), and values describing each video source. Video source description contains, again, the video source "token", frame rate and the resolution:

```
"video_sources": {
  "token_1": {
    "token": "token_1",
    "framerate": INT,
    "resolution": [INT, INT]
  },
  ...,
  "token_K": {
    "token": "token_K",
    "framerate": INT,
    "resolution": [INT, INT]
  }
}
```

A typical ONVIF video camera has one video source. The information under the `video_sources` element represents an excerpt from the SOAP response to GetVideoSources call [9].

The `profiles` member is an associative array with keys equal to profiles' "tokens", and values describing each profile:

```
"profiles": {
  "token_1": {
    "token": "token_1",
    "name": STRING,
    "fixed": BOOLEAN,
    "video": {
      "source": "videoSourceToken_M_1",
      "bounds": [ INT, INT, INT, INT ],
      "codec": STRING,
```

```
      "quality": FLOAT,
      "resolution": [ INT, INT ],
    }
  },
  ...,
  "token_K": {
    "token": "token_K",
    "name": STRING,
    "fixed": BOOLEAN,
    "video": {
      "source": "videoSourceToken_M_K",
      "bounds": [ INT, INT, INT, INT ],
      "codec": STRING,
      "quality": FLOAT,
      "resolution": [ INT, INT ],
    }
  }
}
```

The profile description contains its `token` again, its human-readable `name`, the `fixed` flag showing whether this profile can be deleted, and the video settings. `video.source` value references the token of the video source that is used by this profile. Rest of values under `video` section concerns video encoder settings: `codec` name (Viinex only supports profiles with `codec` equal to `"H264"`), output video `quality` in relative units, `bounds` which is rectangle ROI on the original video source that is taken by video encoder to produce the stream, and `resolution` of the resulting video stream.

The information under the `profiles` element represents an excerpt from the SOAP response to GetProfiles call [9].

### 3.3.7 Discover connected raw video sources

#### Request purpose

Obtain the list of raw video sources visible by the server being asked.

#### Request URL and applicable methods

`GET http://servername:port/v1/env/`*`rawvideo`*

#### Request parameters

none

#### Response syntax

An array of JSON objects, each having three fields: `name`, `address` and `capabilities`:

```
[
  {
    "name": STRING,
    "address": STRING,
    "capabilities": [
      OBJECT_1,
      ...,
      OBJECT_N ]
  },
  ...
]
```

`name` property contains a human-readable name of the device reported by operating system (the driver). `address` is the path of the device, as it should be used in respective property of configuration section of `rawvideo` object. The `capabilities` array is the list of modes reported to be appropriate for the device. Each element of this list is ready to be used as the value for `mode` property of `rawvideo` object configuration, with the exception that elements in the `capabilities` list never contain optional parameters; they only specify manatory values, particularly: `pin, colorspace, framerate` and `size`. For more information see section 2.8.2.

**Response example**

```
[
  {
    "name": "Behold TV Columbus: A\/V Capture [Slot 1]",
    "address":"\\\\?\\pci#ven_1131&dev_7133&subsys_52010000&rev_f0#5&\
              2b491bae&0&0000f0#{65e8773d-8f56-11d0-a3b9-00a0c9223196}\
                                \\{bbefb6c7-2fc4-4139-bb8b-a58bba724083}",
    "capabilities": [
      {
        "pin": "2",
        "colorspace": "YUY2",
        "framerate": 25,
        "size": [704,576]
      },
      {
        "pin": "2",
        "colorspace": "UYVY",
        "framerate": 25,
        "size": [704,576]
      },
      {
        "pin": "2",
        "colorspace": "RGB",
        "bpp": 24,
        "planes": 1,
        "framerate": 25,
        "size": [704,576]
      },
      {
        "pin": "3",
```

```
        "colorspace": "YUY2",
        "framerate": 25,
        "size": [704,576]
      },
      {
        "pin": "3",
        "colorspace": "RGB",
        "bpp": 16,
        "planes": 1,
        "framerate": 25,
        "size": [704,576]
      }
    ]
  },
  {
    "name": "Microsoft Corp. LifeCam HD-3000",
    "address": "\\\\?\\usb#vid_045e&pid_0779&mi_00#6&145ddd63&0&0000#\
                      {65e8773d-8f56-11d0-a3b9-00a0c9223196}\\global",
    "capabilities": [
      {
        "pin": "0",
        "colorspace": "YUY2",
        "framerate": 30,
        "size": [640,480]
      },
      {
        "pin": "0",
        "colorspace": "YUY2",
        "framerate": 30,
        "size": [160,120]
      },
      {
        "pin": "0",
        "colorspace": "YUY2",
        "framerate": 10,
        "size": [1280,800]
      }
    ]
  }
]
```

In the above example, two DirectShow devices were discovered, one TV tuner and one USB videocamera. A number of operation modes is reported for each device.

## 3.4 Video source

### 3.4.1 Status information

#### Request purpose

Obtain the status information for live video source

#### Request URL and applicable methods

`GET http://servername:port/v1/svc/`*`videosourceN`*

#### Request parameters

none

#### Response syntax

JSON object:

```
{
  "last_frame": TIMESTAMP,
  "resolution": [INT, INT],
  "bitrate": INT
}
```

where `last_frame` – a timestamp of the last frame received from the source (may be used to determine whether the video data is actually transferred to viinex 3.4; `resolution` – a pair of numbers indicating width and height of an image (parsed from video stream); `bitrate` – video stream bit rate, estimated using data currently buffered for HLS.

#### Response example

```
{
  "last_frame": "2016-11-11T00:20:23.292Z",
  "resolution": [1280, 960],
  "bitrate": 4194304
}
```

### 3.4.2 Live stream

#### Request purpose

Obtain the HLS playlist for live video from a video source

#### Request URL and applicable methods

`GET http://servername:port/v1/svc/`*`videosourceN/stream`*

#### Request parameters

none

#### Response syntax

M3U8 playlist containing URLs for obtaining video segments. Network clients supporting playback of content delivered via HLS (such as Microsoft Edge or Apple Safari web browsers) can play requested stream without additional requirements.

#### Response example

Request:

```
GET http://192.168.0.70:8880/v1/svc/cam2/stream
```

Response:

```
#EXTM3U
#EXT-X-VERSION:3
#EXT-X-TARGETDURATION:9.93
#EXT-X-MEDIA-SEQUENCE:42

#EXTINF:9.92,
stream/ts/0001bcf4000000000000000001598e8c060c000001598e8c0d8b.ts
#EXTINF:9.92,
stream/ts/0001bcf5000000000000000001598e8c0ddc000001598e8c155b.ts
#EXTINF:9.92,
stream/ts/0001bcf6000000000000000001598e8c15ac000001598e8c1d2b.ts
```

#### Remarks

Note that viinex 3.4 does not implement HLS streaming of any other media except for H.264 video. That is, H.265 and audio tracks within a media stream are not supported. It is suggested that WebRTC or RTSP is used for re-streaming of these media formats.

## 3.5 Video archive

Assuming the video archive component has the name `storageN` in viinex 3.4 configuration, the following URLs become available when video archive is published in the web server:

### 3.5.1 Status and statistics

#### Request purpose

Acquire general statistics data for video archive

#### Request URL and applicable methods

`GET http://servername:port/v1/svc/`*`storageN`*

#### Request parameters

none

#### Response syntax

JSON object:

```
{
  "disk_usage": INTEGER,
  "disk_free_space": INTEGER,
  "contexts": {
    "videosource1": {
      "time_boundaries": [TIMESTAMP, TIMESTAMP],
      "disk_usage": INTEGER
    },
    ...,
    "videosourceN": {
      "time_boundaries": [TIMESTAMP, TIMESTAMP],
      "disk_usage": INTEGER
    }
  }
}
```

where `disk_usage` – disk space, in bytes, used by stored video data (in total and for each video source); `disk_free_space` – disk space, in bytes, free on the volume that video archive uses; `contexts` – list of attached video sources; `videosource1`…`videosourceN` – names of attached video sources. These names match the names of corresponding objects in viinex 3.4 configuration; `time_boundaries` – array of two elements containing timestamps of oldest video fragment's beginning and most recent video fragment's end for that context (video source).

#### Response example

```
{ "disk_usage":476881111724,
  "disk_free_space":119142789120,
  "contexts":{
  "cam1":{
```

```
    "time_boundaries":["2016-11-01T09:00:34.024Z",
                       "2016-11-11T11:25:18.917Z"],
    "disk_usage":303687225548},
  "cam2":{
    "time_boundaries":["2016-11-01T09:01:31.563Z",
                       "2016-11-11T11:26:00.216Z"],
    "disk_usage":173193886176}
  }
}
```

### 3.5.2 Archive contents

#### Request purpose

Obtain detailed information on video archive contents for specific context with identifier `videosourceM`.

#### Request URL and applicable methods

`GET http://servername:port/v1/svc/`*`storageN/videsourceM`*

#### Request parameters

none

#### Response syntax

JSON object:

```
{ "time_boundaries": [TIMESTAMP,TIMESTAMP],
  "disk_usage": INTEGER,
  "timeline": [
    [TIMESTAMP,TIMESTAMP],
    ...,
    [TIMESTAMP,TIMESTAMP]
  ]
}
```

where `time_boundaries` – a pair of timestamps of oldest video record beginning and most recent video record ending; `dist_usage` – disk space, in bytes, used by video data from that video source; `timeline` – an array (sorted in ascending order) of pairs of timestamps, each describing a segment of continuous video record.

#### Response example

```
{ "time_boundaries":["2016-11-01T09:51:31.333Z",
```

```
                    "2016-11-11T12:08:03.856Z"],
  "disk_usage":173074472533,
  "timeline":[
    ["2016-11-01T09:51:31.333Z","2016-11-01T21:03:11.673Z"],
    ["2016-11-01T21:06:09.008Z","2016-11-01T21:31:28.428Z"],
    ...
    ["2016-11-11T00:12:54.954Z","2016-11-11T00:20:23.292Z"],
    ["2016-11-11T01:05:12.396Z","2016-11-11T12:08:03.856Z"]
  ]
}
```

### 3.5.3 Disk usage for a specific time interval

#### Request purpose

Obtain the disk usage for a specific video archive `storageN` and specific video source `videosourceM` on a given time interval.

#### Request URL and applicable methods

`GET http://servername:port/v1/svc/`*`storageN/videsourceM`*`/du`

The last `/du` part of the URL comes from the name of UNIX utility `du`, whose purpose is to estimate the disk usage for chosen files or directories.

#### Request parameters

Time interval boundaries, `?begin=TIMESTAMP&end=TIMESTAMP`

#### Response syntax

The HTTP response body containse the JSON object of the form:

```
{ "disk_usage": INTEGER }
```

#### Response example

```
$ curl "http://demo.viinex.com/v1/svc/stor0/camViinexPond1/du?\
begin=2019-02-28T05:00:00&end=2019-02-28T17:00:00"
{"disk_usage":475048275}
```

#### Remarks

The figure given as the result of this call is a rough estimate of disk usage for specified time interval. Only the size of media files that overlap with requested time interval is taken into account, but not the structure of media data within those files.

### 3.5.4 Overall disk usage for a specific time interval

#### Request purpose

Obtain the disk usage for a specific video archive `storageN` on a given time interval.

#### Request URL and applicable methods

`GET http://servername:port/v1/svc/`*`storageN`*

#### Request parameters

Time interval boundaries, `?begin=TIMESTAMP&end=TIMESTAMP`

#### Response syntax

The HTTP response body containse the JSON object of the form:

```
{
  "disk_usage": INTEGER,
  "contexts": {
    "videosource1": {
      "disk_usage": INTEGER
    },
    ...,
    "videosourceN": {
      "disk_usage": INTEGER
    }
  }
}
```

Note that this syntax is the partial form of the reponse described in section 3.5.1. These two calls are distinguished with the presence/absence of `begin` and `end` call parameters in their request URLs.

#### Response example

```
$ curl "http://demo.viinex.com/v1/svc/stor0?begin=2019-02-28T05:00:00\
&end=2019-02-28T17:00:00"
{"contexts":{"camViinexPond1":{"disk_usage":475048275}},"disk_usage":475048275}
```

### 3.5.5 Media export

#### Request purpose

Export video data for specific context with identifier `videosourceM`.

### Request URL and applicable methods

`GET http://servername:port/v1/svc/storageN/videsourceM/export`

### Request parameters

`begin` – mandatory parameter, requested timestamp to begin video data export from. Integer number of milliseconds elapsed since UNIX epoch (1970-01-01 00:00:00.000 UTC). Note that actual export may be performed from different point, which is selected according to the following rules:

- if there is no continuous video fragment containing the requested `begin` point, export is performed from the nearest video fragment starting after that requested point;
- if there is a continuous video fragment containing requested `begin` point, export actually from start of the GOP which contains requested `begin` point.

That is, if `begin` points no a non-IDR frame, the export will begin from nearest IDR frame preceding that point (there will be some amount of pre-roll frames, necessary to correctly decode video frame at requested `begin` point).

`end` – mandatory parameter, requested timestamp to end video data export at. Integer number of milliseconds elapsed since UNIX epoch.

`format` – optional parameter, which sets the desired output format (container) for video. There are three possible values for this parameters: `isom` for export in ISO-14496 part 12 format [1] (also known as MP4); `ts` for ISO-13818 part 1 format [2] (also known as MPEG TS), and `raw` for obtaining raw H.264 stream with separator NAL units, without container (and as consequence with no timing information). The default behavior (if no value is specified for `format` parameter) is to export data in MP4 (`isom`) format.

`timebase` – optional parameter which is applicable only for `format=ts` case, which allows to set desired time base in the stream. If set, the presentation timestamps set for each frame in the TS, will be counted exactly from requested time base. This allows for precise positioning in exported video fragments.

### Response syntax

A binary content in MP4, MPEG TS or raw H264 format, depending on the value of "format" parameter. "Content-Disposition" header is set in HTTP response to set recommended file name and extension.

### Response example

```
http://192.168.0.70:8880/v1/svc/stor0/cam2/export?
    begin=1478545200000&end=1478545800000&
    format=ts&timebase=1478545200000
```

Export video from video source `cam2` attached to video archive `stor0`, starting at 07 Nov 2016 19:00:00 GMT (1478545200000), finishing at 07 Nov 2016 19:10:00 GMT. Create a MPEG2 Transport stream, marking timestamps so that PCR wraparound occurs exactly at 07 Nov 2016 19:00:00 GMT (so that if there's preceding IDR and non-IDR frames before non-IDR frame at 07 Nov 2016 19:00:00 GMT, such "pre-roll" frames will get negative timestamp values).

#### Remarks

Note that viinex 3.4 does not implement exporting in TS container for any media except for H.264 video. For exporting of H.265 video and audio tracks it is suggested that MP4 container is used.

### 3.5.6 Media playback

#### Request purpose

Obtain playlist for HTTP Live Streaming [3] of video data from video source `videosourceM` being stored in video archive `storageN`.

#### Request URL and applicable methods

`GET http://servername:port/v1/svc/`*`storageN/videsourceM/stream`*

#### Request parameters

`begin` – mandatory parameter, requested timestamp to begin video data export from. Integer number of milliseconds elapsed since UNIX epoch (1970-01-01 00:00:00.000 UTC).

`end` – mandatory parameter, requested timestamp to end video data export at. Integer number of milliseconds elapsed since UNIX epoch.

#### Response syntax

M3U8 playlist containing URLs for obtaining video segments. Network clients supporting playback of content delivered via HLS (such as Microsoft Edge or Apple Safari web browsers) can play specified video fragment (from begin to end) without additional requirements.

#### Response example

Request:

```
GET http://192.168.0.70:8880/v1/svc/stor0/cam2/stream
     ?begin=1478545200000&end=1478545300000
```

Response:

```
#EXTM3U
#EXT-X-PLAYLIST-TYPE:VOD
#EXT-X-VERSION:3
#EXT-X-TARGETDURATION:9.93
#EXT-X-MEDIA-SEQUENCE:0

#EXTINF:9.92,
stream/ts?zero=1478545199866&begin=1478545199866&end=1478545209786
#EXTINF:9.92,
stream/ts?zero=1478545199866&begin=1478545209866&end=1478545219786
...
#EXTINF:9.92,
stream/ts?zero=1478545199866&begin=1478545289876&end=1478545299796
#EXTINF:2.02,
stream/ts?zero=1478545199866&begin=1478545299876&end=1478545301796
#EXT-X-ENDLIST
```

#### Remarks

Note that viinex 3.4 does not implement HLS streaming of any other media except for H.264 video. That is, H.265 and audio tracks within a media stream are not supported. It is suggested that WebRTC or RTSP is used for re-streaming of these media formats.

### 3.5.7 Remove records from video archive

#### Request purpose

Control which records should be removed from video archive.

#### Request URL and applicable methods

`DELETE http://servername:port/v1/svc`*`/storageN/videsourceM`*
`?begin=TIMESTAMP&end=TIMESTAMP`

#### Request parameters

`begin` – mandatory parameter, requested timestamp to begin video data removal from. Integer number of milliseconds elapsed since UNIX epoch (1970-01-01 00:00:00.000 UTC).

`end` – mandatory parameter, requested timestamp to end video data removal at. Integer number of milliseconds elapsed since UNIX epoch.

#### Response syntax

The `DELETE` call makes video archive implementation to remove video recordings specified by the time interval.

#### Response example

Request:

```
DELETE http://192.168.0.70:8880/v1/svc/stor0/cam2
     ?begin=1478545200000&end=1478545300000
```

means a request to remove video data in video archive `stor0` for channel `cam2`, starting from 2016-11-07 19:00:00 UTC (1478545200000), ending on 2016-11-07 19:01:40 UTC (1478545300000).

#### Remarks

As the data in video storage is organized in a number of relatively small video files, of default size about 16 MB each. The video archive implementation in Viinex does not modify those video files after they are formed. The video data removal operation is actually performed as the removal of all video files that have non-empty intersection with selected time interval [begin, end].

The `DELETE` API method is disabled by default. In order to enable it, the `allow_removal` property should be set to `true` in the configuration section for respective instance of video archive. For detailed syntax see section 2.2.1.

## 3.6 Recording controller

### 3.6.1 Status information

#### Request purpose

Obtain the information on recording controller `recctlN`, that is – attached video sources, video storage, and current status.

#### Request URL and applicable methods

`GET http://servername:port/v1/svc/`*`recctlN`*

#### Request parameters

none

#### Response syntax

JSON object:

```
{
  "sources": ["source1", ..., "sourceM"],
```

```
  "storage": "storageK",
  "status": "on" | "off"
}
```

where

`source1`…`sourceM` are identifiers of video sources attached to this recording controller (according to viinex 3.4 configuration);

`storageK` is an identifier of video storage this recording controller is attached to;

`status` – current status of the recording controller (`on` if controller currently transfers the data to the storage, `off` if it does not).

#### Response example

```
{"status":"on",
 "sources":["cam2","cam1"],
 "storage":"stor0"}
```

### 3.6.2 Change recording status

#### Request purpose

Change the status of recording controller `recctlN`.

#### Request URL and applicable methods

`POST http://servername:port/v1/svc`*`/recctlN/start`*

(to set recording "on"), or

`POST http://servername:port/v1/svc`*`/recctlN/stop`*

(to set recording "off").

#### Request parameters

none

#### Preconditions

In order for the recording controller status to be changed, it should initially be in the status that is opposite to the one that was requested. In other words, to start recording, the recording should initially be stopped, and vice versa.

#### Response syntax

Upon successful scheduling of status change command, HTTP status 200 is returned. The JSON body returned in successful response to this request should be ignored by the caller. For backwards compatibility, the response holds a JSON object

```
{ "status": "on" | "off" }
```

indicating a new status, or HTTP error with code 412 if precondition was not met. However this return type is deprecated and will be replaced with an empty JSON object `[]` in the future. If the caller needs the actual status of recording controller, an explicit request 3.6.1 should be used.

#### Response example

```
{"status":"on"} | []
```

#### Remarks

The API given above is straightforward, however it can be effectively used only in assumption that there's only one client to the Recording controller object. If there are multiple clients to the controller, they are responsible for resolving the conflicts of multiple concurrent "start recording" and "stop recording" requests.

### 3.6.3 Flush accumulated video data to disk

#### Request purpose

Flush all of the data accumulated in memory so far to form a new file (video archive fragment) on the disk.

#### Request URL and applicable methods

`POST http://servername:port/v1/svc/recctlN/flush`

#### Request parameters

none

#### Remarks

The flush call forces the video archive that is linked with the recording controller to immediately complete the "current" video fragment.

This call helps to ensure that a) video data accumulated so far by the recording controller is flushed to the disk, so that in case of power failure that data would be available; and b) that video data is available in the video storage using its API.

## 3.7 Managed replication

This section describes the HTTP API exposed by a replication sink in managed mode. Such mode assumes that a sink accepts replication tasks and executes them. Respectively, the programming interface is organized as a CRUD API for managing the replication tasks.

### 3.7.1 Enqueue a new replication task

#### Request purpose

Schedule a new replication task for managed replication sink `replsinkN`.

#### Request URL and applicable methods

`PUT http://servername:port/v1/svc/`*`replsinkN`*`/task/`*`TASK_ID`*

#### Request parameters

TASK_ID – an identifier of the new replication task (unique string). Chosen by client. Request body – a JSON object desribing the replication task, see below syntax details.

#### Response syntax

The client should generate a task identifier – a unique string to distinguish between other replication tasks that might be queued in the replication sink. This task identifier should be the last part of URI path in this call.

The request body for this method should be a JSON object having the following form:

```
{
  "origin": { "type": "vmschan",
              "name": STRING,
              "speed": NUMBER }
          | { "type": "mediafile",
              "path": STRING }
          | { "type": "rtsp",
              "url": STRING,
              "auth": [STRING, STRING],
              "transport": [ "tcp" | "udp" ],
              "speed": NUMBER },
  "meta": JSON,
```

```
  "rem": "For replication task of type rtsp:",
  "url": STRING,
  "auth": [STRING, STRING],
  "transport": ["tcp"|"udp"],


  "rem": "For replication task of type vmschan:",
  "name": STRING,

  "rem": "For replication task of type mediafile:",
  "path": STRING,

  "speed": NUMBER,
  "channel": STRING,
  "begin": TIMESTAMP,
  "end": TIMESTAMP | null,
  "suspend": BOOLEAN
}
```

The replication task represents the description of a source of video, and the instructions on how the video should be copied to the destination storage.

The `type` property specifies the type of replication source. Value "rtsp" means that video data has to be replicated from some RTSP server. In that case, the properties `url`, `auth` and `transport` should be also specified. The meaning of these properties matches that for respective properties for the RTSP video source configuration, as described in section 2.1.1.

Another possible value for the `type` property is ``vmschan''. This special value means that replication source is a third party video management system, which is configured and connected within this instance of viinex 3.4 as described in section 2.1.5. In this case, the property `name` should be specified in the replication task; this property should be equal to the name of object of type `vmschan`, which represents a video channel from a third-party VMS.

The third possible value for the `type` property is ``mediafile''. This value means that a fragment of video data should be read from a local media file and stored in viinex 3.4 video archive. For this type of replication task, the property ``path'' should be specified; it should contain the path to a source media file on a local filesystem (on the computer where viinex 3.4 runs).

The `meta` property may contain an arbitrary JSON value and can be used by client software to store additional identification or other information for the replication task.

Numeric property `speed` specifies the data rate at which the video frames should be received from the replication source. For instance, specifying `"speed":8` means that viinex 3.4 would request data transmission rate of 8, which would result in 8 times faster replication in comparison to the normal playback speed. (That is, 10 minutes of video would replicate in 1 minute and 15 seconds at speed 8).

The property `channel` specifies the name of a video channel in the video archive where the video data should be stored. The properties `begin` and `end` specify the first and the last timestamp of the video fragment that should be replicated, – how it should be placed in the target video storage.

The `suspend` flag, when set to `true`, indicates that the replication task should not be started immediately, but rather it should be placed in the queue in the "paused", or "suspended" state.

This state can be later changed by means of respective command, see section 3.7.3 for details.

#### Response example

```
$ cat repltask1.json
{
    "origin": {
        "type": "rtsp",
        "url": "rtsp://192.168.0.71:554/stor0/cam2?
begin=2017-06-21T08:36:19.365Z&end=2017-06-21T08:37:11.327Z"
    },
    "meta": {},
    "channel": "cam2",
    "begin": "2017-06-21T08:36:19.365Z",
    "end": "2017-06-21T08:37:11.327Z",
    "suspend": false,
    "speed":4
}

$ curl 'http://localhost:8880/v1/svc/replsink1/task/1' -X PUT \
  --data-binary @repltask1.json
```

The second command, given the file repltask1.json is present with the given content, allows one to replicate video from an RTSP source (in this case this is an RTSP server brought up in another instance of viinex 3.4). Video data is replicated at speed of 4 times faster than normal playback speed.

### 3.7.2 Get information on replication task

#### Request purpose

Get information on replication task and its status

#### Request URL and applicable methods

`GET http://servername:port/v1/svc/`*`replsinkN`*`/task/`*`TASK_ID`*

#### Request parameters

TASK_ID – an identifier of the the replication task to retreive information for.

#### Response syntax

Upon success, the response to this query is a JSON object of the following form:

```
{
```

```
  "id": STRING,
  "meta": JSON,
  "origin": { "type":"rtsp", "url":STRING }
          | { "type":"vmschan", "name":STRING },
  "channel": STRING,
  "begin": TIMESTAMP,
  "end": TIMESTAMP,
  "suspend": BOOLEAN,

  "status": "queued" | "running" | "suspended"
          | "completed" | "cancelled" | "failed",
  "error": STRING,
  "status_changed": TIMESTAMP,
  "time_elapsed": NUMBER,
  "last_frame": TIMESTAMP,
  "bytes_received": NUMBER
}
```

The properties `id`, `meta`, `origin`, `channel`, `begin`, `end` and `suspend` do not change over time and repeat the information provided by a client at the moment when the replication task was created. Their meaning is described in the previous section, except for the property `origin` which combines the options for and arbitrary RTSP video source and a video channel from a thirdparty VMS.

The property `status` indicate the current status of the replication task. For discussion on statuses, see the next section. The property `error` may contain a stringified error message for the task in status `failed`. The `status_changed` property indicates the moment, according to server's system clock, when the status of the task has changed.

The `time_elapsed` property contains the number of seconds that was spent on the execution of this task, except the time the task is in `running` status since the most recent status change. This means that a) only the time when the task is in status `running` is taken into account, but if the task is currently running, this time does not change with every request – instead the client may compute current running time comparing wall clock with the value of `status_changed` property. For final statuses (``completed'', ``cancelled'', ``failed'') the field `time_elapsed` show the proper total number of seconds which this task was running (but not queued or suspended).

The `last_frame` property holds the timestamp of the last frame that was received. It could be `null`, if no frames were received yet. For running tasks, when data retrieval is in progress, this field takes the value in interval of `[begin, end]`. After the timestamp of last received frame exceeds the `end` timestamp, such task is considered completed, even if the data source continues to send more data. By means of this value a relative progress for execution of the replication task may be estimated.

The `bytes_received` property holds the number of bytes that were acquired from the replication source to the time of request. As a rule the total size of footage to be replicated, in bytes, is unknown in advance, therefore this property can only be used for statistics and/or general information. For the tasks where no video data has arrived yet from replication source, this property is `null`.

#### Response example

```
$ curl 'http://localhost:8880/v1/svc/replsink1/task/1' -X GET
{
  "id":"1",
  "meta":{},
  "origin":{"type":"rtsp","url":"rtsp://127.0.0.1:554/stor0/cam2
?begin=2017-06-21T08:36:19.365Z&end=2017-06-21T08:37:11.327Z"},
  "begin":"2017-06-21T08:36:19.365Z",
  "end":"2017-06-21T08:37:11.327Z",
  "channel":"cam2",
  "suspend":false,

  "time_elapsed":1.9881,
  "bytes_received":12975144,
  "status":"completed",
  "last_frame":"2017-06-21T08:37:11.303311111083Z",
  "status_changed":"2019-09-25T15:52:04.0027018Z"
}
```

This example shows the status for a successfully completed replicaiton task. 12975144 bytes were replicated in 1.99 seconds into channel `cam2` of the video storage.

### 3.7.3 Manage status of replication task

#### Request purpose

Change the status of replication task: put it on hold, or return to the queue, or cancel

#### Request URL and applicable methods

`POST http://servername:port/v1/svc/`*`replsinkN`*`/task/`*`TASK_ID`*

#### Request parameters

TASK_ID – an identifier of the the replication task to retreive information for. Request body should be a JSON object describing the action for the replication task, see below.

#### Response syntax

The HTTP body of this request should be a JSON object of the form

```
{ "command": "suspend" | "resume" | "cancel" }
```

The task may reside in one of six statuses: `queued`, `running`, `suspended`, `completed`, `cancelled` and `failed`. First three statuses are transitional. The task is created in status `queued` (or `suspended`, if respective flag is set to `true` upon task creation). When a free

replication worker is encountered, it finds a queued task and starts its execution; at this moment the `queued` task status is changed to `running`. A running task can fail or complete successfully, – in that case it would transit to the status `failed` or `completed`. While running, a task can be suspended or canceled. A task which is suspended, can also be canceled; or it can be queued again (which means that over time, once a free worker is found for that task, it would become `running`).

This HTTP call serves for the purpose of changing the task status. Respective values of the `command` property mean: `suspend` a task – to pause its execution. Note that in general case it's not always possible to correctly suspend a task, because the connection to the replication source would not necessarily be able to preserve the context. It's safe though to suspend a task which was `queued`, but, due to the lack of free workers in replication sink, was not yet put into `running` status.

To `resume` a task means, respectively, to queue the task which was previously suspended or which was created with `suspended` flag set to `true`. After the task is resumed, it takes the last place in the queue. That is, in the situation when there is a lack of free replication workers and a number of queued tasks, the suspension and further resuming of a task effectively places this task to the end of the queue.

When the `cancel` command is issued for a task, the data transmission for such task is terminated, and the task is marked as "cancelled". A cancelled task cannot be started over, paused or queued again; it is one of final states. Cancelled task can only be deleted (see section 3.7.4).

#### Remarks

### 3.7.4 Remove a replication task

#### Request purpose

Remove replication task which is already finalized from the queue of replication sink

#### Request URL and applicable methods

`DELETE http://servername:port/v1/svc/`*`replsinkN`*`/task/`*`TASK_ID`*

#### Request parameters

TASK_ID – an identifier of the the replication task to be removed.

#### Remarks

Only the task which is in one of the final states, – that is, completed, or cancelled, or failed, – can be removed. An attempt to remove a running, queued or suspended task would result in an error. Therefore to forcibly terminate a task, one needs to cancel it first, and then to remove this task from the queue.

Also note that the queue is not persistent. Finalized tasks stay in the memory of replicaiton sink just for API consistence. If an instance of viinex 3.4 is restarted, all previously queued

replication tasks would be lost.

### 3.7.5 Enumerate all replication tasks

#### Request purpose

Get all tasks from the managed replication sink

#### Request URL and applicable methods

`GET http://servername:port/v1/svc/`*`replsinkN`*`/task`

#### Request parameters

none

#### Response syntax

Upon success, a JSON dictionary is returned, whose keys are identifiers of the tasks, and values are the status information objects for respective tasks, as described in 3.7.2:

```
{
  TASK_ID_1: JSON,
  TASK_ID_2: JSON,
  ...
  TASK_ID_N: JSON
}
```

For instance, the output to this HTTP request could be as follows:

```
$ curl 'http://localhost:8880/v1/svc/replsink1/task' -X GET
{
  "1":{"id":"1","origin":..., ...,"bytes_received":12975144},
  "2":{"id":"2","origin":..., ...,"bytes_received":null},
}
```

### 3.7.6 Get the timeline from a VMS channel

#### Request purpose

Acquire the information on video recordings stored in a third-party video archive for a specific VMS channel

#### Request URL and applicable methods

`GET http://servername:port/v1/svc/`*`vmschanN`*`/timeline`

#### Request parameters

Optional parameters `begin` and `end`

#### Response syntax

The `timeline` call allows a client to acquire the timeline of an external video archive for specific video channel. Optional parameters `begin` and `end`, both of timestamp type, may be used to indicate the interval of interest. Depending on undelying VMS implementation, this may save resources and speed up the execution of this request.

Upon success, the response to this request is a JSON array containing timeline intervals; each interval is encoded as a JSON array of exactly 2 elements:

```
[ [TIMESTAMP_1_BEGIN, TIMESTAMP_1_END],
  [TIMESTAMP_2_BEGIN, TIMESTAMP_2_END],
  ...
  [TIMESTAMP_N_BEGIN, TIMESTAMP_N_END] ]
```

Note that this information can be lengthy to obtain, depending on the underlying VMS implementation. viinex 3.4 does not perform any caching for this call.

## 3.8 Snapshots

### 3.8.1 Get a snapshot from the snapshot source

#### Request purpose

Get an image containing the snapshot from the specified video source which implements snapshots.

#### Request URL and applicable methods

`GET http://servername:port/v1/svc/`*`sourceN/snapshot`*

to acquire a snapshot from a live video source, or

`GET http://servername:port/v1/svc/`*`storageM/sourceN/snapshot`*

to extract a single frame from a video archive

### Request parameters

All parameters are optional: `timestamp`, `cached`, `scale`, `width`, `height`, `roi`. See remarks for details.

### Response example

```
curl http://localhost:8880/v1/svc/raw0/snapshot
```

Gets the JPEG snapshot image from raw video source `raw0`.

```
curl 'http://localhost:8880/v1/svc/stor0/cam2/snapshot?scale=3&cached=10'
```

Gets the first frame of 10th most recent archive fragment for source `cam2` within video archive `stor0`. Downscale it 3 times in each dimension.

```
curl 'http://localhost:8880/v1/svc/stor0/cam1/snapshot?roi=(0.5,0.5,0.7,0.8)\
&height=50&timestamp=1505678400000'
```

Extract the frame with timestamp 1505678400000 (September 17, 2017 8:00:00 PM UTC) for video source `cam1` within video archive `stor0`. Crop the image to the ROI with geometry $(0.5, 0.5, 0.7, 0.8)$; scale the result to have height of 50 pixels, preserving aspect ratio.

### Remarks

There are two types of requirements that may be established when the call for acquiring a snapshot is issued: these are temporal and spatial requirements.

Temporal requirements are most useful when a frame from the video archive is requested. Two mutually exclusive parameters may set the temporal requirement: `timestamp` or `cached`. The `timestamp` parameter should be a string in ISO 8601 date and time format[1], or an integer number equal to the number of milliseconds elapsed since UNIX epoch (midinght of January 1st, 1970) till the moment when the video frame that should be extracted was shot. If this parameter is given, viinex 3.4 extracts the minimal video fragment which includes requested moment, finds the requested video frame, and returns it. Note that if the requested moment is absent in the video archive, viinex 3.4 returns an error. The `cached=N` parameter, when used in request to a video archive, instructs viinex 3.4 to extract a first frame from Nth most recent video fragment written so far to the video archive. This is a cheap and easy way to produce an overview of a video archive contents. In particular, the parameter `cached=0` which is equivalent to not specifying temporal-related parameters at all, – returns the first frame from the most recent video fragment written to the archive.

When given in the snapshot request to a live source, the only allowed temporal-related requirements are the either the absence of parameters, or the presence of parameter `cached=0`. The difference between the two cases is how the last obtained and cached snapshot for that live

[1] In particular, the format should be of the form `"YYYY-mm-ddTHH:MM:SS.fffZ"`, where `YYYY` is the gregorian year number, four digits; `mm` is 1-based month number, two digits; `dd` – 1-based day number in month, two digits; `HH` – hour from 0 to 23, two digits; `MM` – minute number from 0 to 59, two digits; `SS` – seconds, from 0 to 59, two digits; `.fff` – optional point and fractional portion of second, from 1 to 6 digits; 'T' and 'Z' are letters, '-' and ':' are dashes and semicolons on their respective fixed positions.

source will be used in this request. When no parameters are given, viinex 3.4 checks how long ago the snapshot for the queried live source was obtained for the last time. If it was obtained within a reasonable period, the cached snapshot is returned. Otherwise, if the previously obtained snapshot is considered "stalled", the new one is acquired from data source, cached, and returned to the client. However, if the parameter `cached=0` is present in the request, the snapshot is obtained from data source unconditionally, no matter how new the snapshot which already present the cache is. Note that this operation typically involves time-consuming I/O to other devices like IP cameras.

Spatial requirements for obtaining a snapshot may be given by means of parameters `scale`, `width/height` and `roi`. All of them are optional; of none is given, the original image is returned. The `roi` parameter may be given independently from other spatial requirements. This parameter should have the form of `(L,T,R,B)` – four comma-separated floating-point numbers, denoting coordinates of the left-top and right-bottom corners of ROI (region of interest) to crop from original image. All that numbers are expected to be in relative units, in the range $[0, 1]$, where 1 means maximum possible value which is equal to original image width for `L,R` or height for `T,B`. For example, consider the parameter `roi=(0.2,0.3,0.55,0.45)`. ROI width and height yields 0.35 and 0.15 respectively. If original image size is $800 \times 600$, then ROI left-top corner is $(160, 180)$, and ROI size is $280 \times 90$ (measured in pixels on the original image).

The `scale=N` parameter instructs viinex 3.4 to downscale the image, reducing its size $N$ times in each dimension (to preserve the aspect ratio). The value $N$ should be an integer.

As an alternative to parameter `scale`, the parameter `width` or `height`, or both, may be given. In the presence of any of that two parameters, viinex 3.4 is instructed to resize the output image so that it has at least specified `width`, or `height`, or both, if both are given. Note that viinex 3.4 preserves the aspect ratio of the image, no matter what spatial-related parameters are specified. If only `width` or only `height` is set, – viinex 3.4 resizes the resulting image to have the specified size in corresponding dimension, – and the second dimension, which is not specified, is chosen so that the aspect ratio is preserved, so that the picture is not distorted. However when both `width` and `height` are set, viinex 3.4 scales the output image so that one of resulting dimenstions exactly matches specified value (width or height), while the second one is greater or equal to the specified value (height or width, respectively). For example, if the original image size is $800 \times 600$, and `scale=4` specified, the resulting image will have the size of $200 \times 150$. If, instead, the parameter `width=500` is specified, the image will be scaled by factor 5/8 times to have width equal to 500, and the corresponding height will be $5 * 600/8 = 375$. However if both `width=320&height=200` are specified, viinex 3.4 would compute corresponding scale factors to match specified width, which is $320/800 = 0.4$, and height, which is $200/600 = 0.(3)$, and choose the bigger one. The scale factor of 0.4 will be applied, and the resulting image will get the size of $320 \times 240$ instead of requested $320 \times 200$.

Note that `width/height` set of parameters is applied "after" the `roi` parameter, that is – corresponding scale factors are computed for the cropped image to match the requested size, not the original one. This should save users effort in practical situations when a cropped image, showing the required ROI only, should fit certain space in a report form or in a GUI.

## 3.9 Overlay

### 3.9.1 Clear overlay

#### Request purpose

Clear the content of overlay number $K$ which is rendered over video for raw video source or video renderer `rawvideoN`.

#### Request URL and applicable methods

`POST http://servername:port/v1/svc/rawvideoN/overlay/K/clear`

#### Request parameters

Overlay number $K$ – a zero-based integer, an index of the overlay to be cleared, according to the configuration of the video source. Can be omitted if there is only one overlay configured.

#### Response example

```
curl http://localhost:8880/v1/svc/raw0/overlay/1/clear -X POST
```

clears the second overlay data of the video source `raw0`.

### 3.9.2 Change overlay bitmap

#### Request purpose

Change the overlay image rendered over video for raw video source or video renderer `rawvideoN`.

#### Request URL and applicable methods

`POST http://servername:port/v1/svc/rawvideoN/overlay/K/bmp`

or

`POST http://servername:port/v1/svc/rawvideoN/overlay/K`

with corresponding `Content-Type` header of value `image/x-ms-bmp`.

#### Request parameters

Overlay number $K$ – a zero-based integer, an index of the overlay to be set, according to the configuration of the video source. Can be omitted if there is only one overlay configured.

The body of request should contain the BMP data to be set as overlay image.

#### Response example

```
$ curl http://localhost:8880/v1/svc/raw0/overlay/0/bmp \
   -X POST --data-binary @overlay.bmp
```

A UNIX command to set the image for the first overlay on the video source `raw0` to the image in file `overlay.bmp`.

### 3.9.3 Change overlay HTML

#### Request purpose

Change the overlay HTML rendered over video for raw video source or video renderer `rawvideoN`.

#### Request URL and applicable methods

`POST http://servername:port/v1/svc`*`/rawvideoN/overlay/K/html`*

or

`POST http://servername:port/v1/svc`*`/rawvideoN/overlay/K`*

with corresponding `Content-Type` header of value `text/html`.

#### Request parameters

Overlay number $K$ – a zero-based integer, an index of the overlay to be set, according to the configuration of the video source. Can be omitted if there is only one overlay configured.

`width` – the width of bitmap to render HTML to.

`height` – the height of bitmap to render HTML to.

`zoom` – zoom to apply to HTML when rendering.

The body of request should contain the HTML data to render and set as overlay image.

#### Response example

```
$ (echo '<body bgcolor="#808080">'; date; echo "</body>") | \
  curl 'http://localhost:8880/v1/svc/raw0/overlay/html?zoom=2&width=300' \
    -X POST --data-binary @-
```

a UNIX command to set the first and only overlay ($K$ parameter is omitted from the path) to current date and time. HTML is rendered to the bitmap of 300 pixels wide; zoom of 2x is applied. The height of bitmap is chosen automatically by HTML renderer. HTML text specifies backgound color "#808080" (gray) which can be used as `colorkey` value `[128,128,128]` in overlay settings, as described in 2.8.4.

## 3.10 Video renderer

When published under the viinex 3.4 web server, the video renderer exposes a number of programming interfaces, namely – a video source interface, a live snapshots source interface, an overlay control interface, and a layout control interface. Respective programming interfaces share the requests syntax with other objects and are described in sections 3.4, 3.8, 3.9 and 3.11.

## 3.11 Layout control

### 3.11.1 Get the names of linked video sources

#### Request purpose

Obtain the sorted list of names video sources linked with this instance of video renderer.

#### Request URL and applicable methods

`GET http://servername:port/v1/svc/`*`rendererN/sources`*

#### Request parameters

None.

#### Response syntax

A JSON array of strings is returned.

#### Response example

```
$ curl -X GET http://localhost:8880/v1/svc/rend0/sources

["cam1","cam2","cam3"]
```

The list of three video sources is returned. In the request for layout control, see section 3.11.2, these video sources should be referred to in the `"input"` parameter by their zero-based index in this array. For instance, `"cam3"` video source should be referred to as `"input":2`.

### 3.11.2 Set the layout for the video renderer

#### Request purpose

Set the image size, background color and viewports parameters for rendering in the resulting

video stream.

### Request URL and applicable methods

`POST http://servername:port/v1/svc/`*`rendererN/layout`*

### Request parameters

Layout description should be passed as POST request body in JSON format.

### Response syntax

The syntax of layout description is given in section 2.8.5. It matches the syntax of layout section in the configuration of video renderer with the exception that it is illegal to specify the background image. For that, one should use an explicit API call, see below.

### Response example

A request can be sent using the CURL utility:

```
curl -X POST http://localhost:8880/v1/svc/renderer0/layout
    --data-binary @lay.json
```

where the file `lay.json` may contain the following text:

```
{
    "size": [1280, 960],
    "background": [99,0,55],
    "viewports": [
        {
            "input": 0,
            "dst": [0.1,0.1,0.7,0.7]
        },
        {
            "input": 1,
            "border": [0,0,255],
            "dst": [0.6,0.05,0.95,0.4]
        },
        {
            "input": 2,
            "border": [0,255,0],
            "dst": [0.6,0.45,0.95,0.9]
        },
        {
            "input": 2,
            "border": [255,0,0],
            "src": [0.1,0.3,0.6,0.6],
```

```
            "dst": [0.05,0.45,0.55,0.9]
        }
    ]
}
```

Here, the video renderer is instructed to set output video size to $1280 \times 960$, set the background color to burgundy, and shows three video sources in four viewports. The fourth viewport shows the same video source as the thrid, but with a "digital zoom": a small ROI is selected on the source video (by means of specifying the `"src": [0.1,0.3,0.6,0.6]` parameter) to be shown in that viewport. Also, each viewport except the first one is enclosed by a border of its own color (blue, green and red for the 2nd, 3rd and 4th viewport respectively).

### 3.11.3 Set the background color or background image

#### Request purpose

Set the background color or the background image for the video renderer[2].

#### Request URL and applicable methods

`POST http://servername:port/v1/svc`*`/rendererN/background`* or
`POST http://servername:port/v1/svc`*`/rendererN/background/color`* or
`POST http://servername:port/v1/svc`*`/rendererN/background/bmp`* or
`POST http://servername:port/v1/svc`*`/rendererN/background/jpeg`*

#### Request parameters

None.

#### Response syntax

The body of the request should contain an image for the background in JPEG or BMP format, or a color for solid background in form of JSON array of 3 integer elements in range $0 \ldots 255$ for red, green and blue component.

In the first case of URL */rendererN/background* the actual type of request is inferred from the MIME type of the body, which should be passed in the `Content-Type` HTTP header of the request. In case of */rendererN/background/color*, */rendererN/background/bmp* and */rendererN/background/jpeg* requests the MIME type header of the request is ignored; the body of the request is expected to be of the format matching the request URL.

[2]This call is mainly to allow setting the background image for the video renderer from the API, taking into account that in the configuration of renderer, the background image is set with the layout. However, since there is a call for setting the layout in the API, and it takes a JSON body, it would be inconvenient to pass the background in the same call, – an explicit API call for setting the background was introduced.

**Response example**

Set the solid color of the background for the video renderer:

```
$ curl -X POST http://localhost:8880/v1/svc/rend0/background/color \
    --data '[66,11,99]'
```

or

```
$ curl -X POST http://localhost:8880/v1/svc/rend0/background \
    -H 'Content-Type: application/json' --data '[66,11,99]'
```

Set the background image from the BMP file:

```
$ curl -X POST http://localhost:8880/v1/svc/rend0/background/bmp \
     --data-binary @background.bmp
```

or

```
$ curl -X POST http://localhost:8880/v1/svc/rend0/background \
    -H 'Content-Type: image/x-ms-bmp' --data-binary @background.bmp
```

Set the background image from the JPEG file:

```
$ curl -X POST http://localhost:8880/v1/svc/rend0/background/jpeg \
     --data-binary @background.jpg
```

or

```
$ curl -X POST http://localhost:8880/v1/svc/rend0/background \
    -H 'Content-Type: image/jpeg' --data-binary @background.jpg
```

### 3.11.4 Set or clear the image for viewports of disconnected video sources

**Request purpose**

Set or clear the image for the viewports on video renderer corresponding to the videosources that are disconnected.

**Request URL and applicable methods**

`POST http://servername:port/v1/svc/`*`rendererN/nosignal`* or
`POST http://servername:port/v1/svc/`*`rendererN/nosignal/bmp`* or
`POST http://servername:port/v1/svc/`*`rendererN/nosignal/jpeg`*

### Request parameters

None.

### Response syntax

The body of the request should contain an image to be displayed in the viewports for disconnected video sources. The image should be in JPEG or BMP format.

In the first case of URL */rendererN/nosignal* the actual type of request is inferred from the MIME type of the body, which should be passed in the `Content-Type` HTTP header of the request. For the first request, the `Content-Type` can be omitted and the request body can be empty; this would indicate that the image for viewports with no signal should be reset. When the image for disconnected video source indication is reset (or not set), the viewports corresponding to such video sources are not displayed on the layout.

In case of */rendererN/nosignal/bmp* and */rendererN/nosignal/jpeg* requests the MIME type header of the request is ignored; the body of the request is expected to be of the format matching the request URL.

### Response example

Set the image for indication of a disconnected video source from the BMP file:

```
$ curl -X POST http://localhost:8880/v1/svc/rend0/nosignal/bmp \
     --data-binary @nosignal.bmp
```

or

```
$ curl -X POST http://localhost:8880/v1/svc/rend0/nosignal \
    -H 'Content-Type: image/x-ms-bmp' --data-binary @nosignal.bmp
```

Set the image for indication of a disconnected video source from the JPEG file:

```
$ curl -X POST http://localhost:8880/v1/svc/rend0/nosignal/jpeg \
     --data-binary @nosignal.jpg
```

or

```
$ curl -X POST http://localhost:8880/v1/svc/rend0/nosignal \
    -H 'Content-Type: image/jpeg' --data-binary @nosignal.jpg
```

Reset (clear) the image for indication of a disconnected video source:

```
$ curl -X POST http://localhost:8880/v1/svc/rend0/nosignal
```

## 3.12 Stream switch

### 3.12.1 Get the names of linked video sources

#### Request purpose

Obtain the sorted list of names video sources linked with this instance of stream switch.

#### Request URL and applicable methods

`GET http://servername:port/v1/svc/`*`streamswitchN/sources`*

#### Request parameters

None.

#### Response syntax

A JSON array of strings is returned.

#### Response example

```
$ curl -X GET http://localhost:8880/v1/svc/vcam1/sources

["cam1","cam2","cam3"]
```

The list of three video sources is returned. In the request for switching the output, see section 3.12.2, the index of a video source identifier in this array should be passed in order to switch the output to respective stream.

### 3.12.2 Switch to a specific stream

#### Request purpose

Set the specific input stream as the output stream of the stream switch object.

#### Request URL and applicable methods

`POST http://servername:port/v1/svc/`*`streamswitchN`*

#### Request parameters

None.

#### Response syntax

The body of the request should have the form of

```
{
    "input": INT
}
```

where the parameter `input` should take the value of zero-based index of requested video source identifier in the sorted list of input video sources linked to this instance of stream switch, as described in section 3.12.1.

Note that this API call is an implementation of an abstract `Updateable` interface described in section 3.17.2.

#### Response example

Given the example from section 3.12.1, the request

```
$ curl -X POST http://localhost:8880/v1/svc/vcam1 --data '{"input":2}'
```

would make the stream switch `vcam1` output the video from the source `cam3`.

## 3.13 PTZ control

PTZ control programming interface provides basic access to the pan-tilt-zoom functionality of an ONVIF device. The API acts merely as a simplifying proxy to that implemented by ONVIF device, as specified in [12]. While the original specification is based on SOAP, viinex 3.4 uses parameters in HTTP requests and JSON where complex data structures are returned to the client. Other differences from the original specification is that viinex 3.4 currently does not support certain features, most noticeable of which are preset tours.

viinex 3.4 does not perform caching of information regarding the PTZ functionality. Neither the clients requests are checked or filtered. All client requests are translated to the device (except the "get node description"), and device's response to each request is translated to respective client.

### 3.13.1 Get the PTZ node description

#### Request purpose

Obtain the name and token of the PTZ node corresponding to the media profile selected at the instance of viinex 3.4 implementation of ONVIF device. Get the numeric limits for motion spaces of the PTZ device.

#### Request URL and applicable methods

`GET http://server:port/v1/svc/onvifN/`*`ptz`*

### Request parameters

none

### Response syntax

The response to this call is a JSON object of the form:

```
{
  "token": STRING,
  "name": STRING,
  "max_presets": INT,
  "home": {
    "supported": BOOLEAN,
    "fixed": BOOLEAN
  },
  "limits": [
    {
      "type": "absolute" | "relative" | "continuous",
      "axis": "PanTilt" | "Zoom",
      "x": [ FLOAT, FLOAT ],
      "y": [ FLOAT, FLOAT ]
    },
    ...
  ]
}
```

Here, `token` is a short string identifying the PTZ node at the ONVIF device. `name` is a human-readable name of that PTZ node. `max_presets` is an indeger that denotes the maximum number of preset positions that the PTZ device is capable of storing. The `home.supported` is a boolean flag meaning whether the "home" position is supported at the PTZ device. The `home.fixed` denotes whether the "home" position is fixed and cannot be changed.

The `limits` is a JSON array containing the structures of three or four elements, `type, axis, x` and, when `axis` equals to the value of `"PanTilt"`, — `y`. The `type` field denotes the motion type supported by the PTZ device.[3] The `axis` field denotes the axis (or axes) along which the motion can be performed. The possible values are `PanTilt` for pan and tilt, and `Zoom` for zoom. The `x` and `y` properties define the limits (maximum and minimum values) for the motion of specified type along specified axis/axes. One range, `x` is specified for the case of one axis, `Zoom`, while two ranges, `x` and `y` are specified for the case of two axes, `PanTilt`.

For more information please refer to the ONVIF PTZ Service specification [12].

[3]There are three motion types distinguished by ONVIF PTZ specification, which are `"absolute"`, `"relative"` and `"continuous"`. The "absolute" motion type means the motion that happens when the device accepts an instruction to move to some position with specific coordinates, no matter what position is current. The coordinates with that command denote the position of PTZ device. Whenever a command to move to an `absolute` position is given, the device shall move to the same position with specified coordinates. The "relative" motion type is the motion that occurs when the device accepts a command to move to a specific distance with respect to its current position. The coordinates with such command denote the displacement of the PTZ device. The "continuous" motion type denotes the motion without a specific destination but with a specific direction and velocity. The coordinates with the command for continuous motion are measured in the units of velocity.

**Response example**

An example of real response to this call is given below:

```
$ curl http://localhost:8880/v1/svc/cam3/ptz
{
  "token": "000",
  "name": "PTZNode_Channel1",
  "max_presets": 80,
  "home": {
    "fixed": false,
    "supported": true
  },
  "limits": [
    {
      "type": "absolute",
      "axis": "PanTilt",
      "x": [-1,1],
      "y": [-1,1]
    },
    {
      "x": [0,1],
      "type": "absolute",
      "axis": "Zoom"
    },
    {
      "type": "relative",
      "axis": "PanTilt",
      "x": [-1,1],
      "y": [-1,1]
    },
    {
      "type": "relative",
      "axis": "Zoom",
      "x": [-1,1]
    },
    {
      "type": "continuous",
      "axis": "PanTilt",
      "x": [-1,1],
      "y": [-1,1]
    },
    {
      "type": "continuous",
      "axis": "Zoom",
      "x": [-1,1]
    }
  ]
}
```

Here, all possible combinations of motion type and axis are suppted. The reported ranges for pan and tilt position, displacement and velocity are all equal to interval $[-1, 1]$ (in their

respective units), as well as the ranges for zoom displacement and velocity. The reported range for zoom range is $[0, 1]$.

### 3.13.2 Get presets

#### Request purpose

Get the tokens and human-readable names for the presets stored in the PTZ device's memory.

#### Request URL and applicable methods

`GET http://server:port/v1/svc/onvifN/ptz/presets`

#### Request parameters

none

#### Response syntax

The reponse for this call is a JSON array of typles (pairs) of strings, of which the first denotes preset's token (an identifier within this PTZ device), while the second is a human-readable name of the preset:

```
[
  [STRING_id_1, STRING_name_1],
  ...
  [STRING_id_k, STRING_name_k],
  ...
]
```

#### Response example

An example of the reponse to the get presets API call is given below:

```
$ curl  http://localhost:8880/v1/svc/cam3/ptz/presets
[["1","Preset1"],["2",""],["5","test preset"]]
```

Here, three presets are defined at the device, with identifiers "1", "2" and "5".

### 3.13.3 Create a preset

#### Request purpose

Define a new preset at the PTZ device, remembering the current position of the device.

#### Request URL and applicable methods

`PUT http://server:port/v1/svc/onvifN/ptz/preset?name=STRING`

#### Request parameters

`name` – an optional URL parameter

#### Response syntax

The `name` is an optional URL string parameter, which defines the human-readable name of the preset.

The response to this request is a JSON object containing one string property:

```
{"token":STRING}
```

The value of `token` property is the identifier given to the new preset by the PTZ device.

#### Response example

An example of creating a preset without a name:

```
$ curl -X PUT  'http://localhost:8880/v1/svc/cam3/ptz/preset'
{"token":"6"}
```

Creating a preset with a name:

```
$ curl -X PUT 'http://localhost:8880/v1/svc/cam3/ptz/preset?name="Front door"'
{"token":"7"}
```

### 3.13.4 Remove a preset

#### Request purpose

Remove an existing preset from the PTZ device's memory

#### Request URL and applicable methods

`DELETE http://server:port/v1/svc/onvifN/ptz/preset/TOKEN`

#### Request parameters

*TOKEN* – an identifier of the preset to remove

#### Response syntax

An identifier (token) of the preset to be removed should be given as a part of the URL path.

The call returns HTTP code 200 and an empty JSON value (`[]`) in the response body on success, or error code 500 with error text in the response body on failure.

#### Response example

```
$ curl -X DELETE  'http://localhost:8880/v1/svc/cam3/ptz/preset/2'
[]
```

Here, the preset with identifier "2" is successfully removed from the ONVIF device configured in viinex 3.4 as `cam3`.

### 3.13.5 Update a preset

#### Request purpose

Update an existing preset, to hold a current position of the PTZ device. Optionally change the name of the preset.

#### Request URL and applicable methods

`POST http://server:port/v1/svc/onvifN`*`/ptz/preset/TOKEN?name=STRING`*

#### Request parameters

*TOKEN* – an identifier of the preset to remove. *name* – an optional parameter, a new name for the preset.

#### Response syntax

An identifier (token) of the preset to be updated should be given as a part of the URL path. The URL parameter `name` can be given to set the new name of the preset.

The call returns HTTP code 200 and an empty JSON value (`[]`) in the response body on success, or error code 500 with error text in the response body on failure.

#### Response example

```
$ curl -X POST  'http://localhost:8880/v1/svc/cam3/ptz/preset/2?name="Reception"'
[]
```

Here, the preset with identifier "2" is successfully updated. The new name of the preset was set.

### 3.13.6 Go to a specified preset

#### Request purpose

Change the PTZ device position to the position stored as a preset with a specified identifier (a token).

#### Request URL and applicable methods

`POST http://server:port/v1/svc/onvifN/ptz/goto/preset/TOKEN`

#### Request parameters

*TOKEN* – an identifier of the preset to recall.

#### Response syntax

An identifier (token) of the preset to recall should be given as a part of the URL path.

The call returns HTTP code 200 and an empty JSON value (`[]`) in the response body on success, or error code 500 with error text in the response body on failure.

#### Response example

```
$ curl -X POST  http://localhost:8880/v1/svc/cam3/ptz/goto/preset/5
[]
```

Here, the device is moved to a position previously stored as preset with identifier "5".

### 3.13.7 Update the "home" position

#### Request purpose

Update the "home" position, to hold a current position of the PTZ device.

#### Request URL and applicable methods

`POST http://server:port/v1/svc/onvifN/ptz/home`

#### Request parameters

none

#### Response syntax

The call returns HTTP code 200 and an empty JSON value (`[]`) in the response body on success, or error code 500 with error text in the response body on failure.

#### Response example

```
$ curl -X POST  'http://localhost:8880/v1/svc/cam3/ptz/home'
[]
```

Here, the "home" position is successfully updated for the ONVIF device configured as `cam3` in a viinex 3.4 instance.

### 3.13.8 Go to the "home" position

#### Request purpose

Change the PTZ device position to the position stored as the "home" position.

#### Request URL and applicable methods

`POST http://server:port/v1/svc/onvifN`*`/ptz/goto/home`*

#### Request parameters

none

#### Response syntax

The call returns HTTP code 200 and an empty JSON value (`[]`) in the response body on success, or error code 500 with error text in the response body on failure.

#### Response example

```
$ curl -X POST  http://localhost:8880/v1/svc/cam3/ptz/goto/home
[]
```

Here, the "home" position is successfully recalled on the ONVIF device configured as `cam3` in a viinex 3.4 instance.

### 3.13.9 Get the coordinates of a current position

#### Request purpose

Obtain the absolute coordinates of current position of the PTZ device.

#### Request URL and applicable methods

`GET http://server:port/v1/svc/onvifN/ptz/position`

#### Request parameters

none

#### Response syntax

The call returns HTTP code 200 and an JSON value of the form

```
[ [ PAN, TILT ], ZOOM ]
```

in the response body on success, or error code 500 with error text in the response body on failure.

Note that for some PTZ devices the `[PAN,TILT]` or the `ZOOM` part may be `null`.

#### Response example

```
$ curl -X GET  http://localhost:8880/v1/svc/cam3/ptz/position
[[-0.22810589,0.9522],0]
```

### 3.13.10 Move the PTZ device

#### Request purpose

Move the PTZ device into an arbitrary position defined by an absolute coordinates or relative displacement, or start continuous motion in a specified direction

#### Request URL and applicable methods

`POST http://server:port/v1/svc/onvifN/ptz/move/TYPE?pan=FLOAT&tilt=FLOAT&zoom=FLOAT`

### Request parameters

*TYPE* – a string taking one of three values: `absolute`, `relative` or `continuous`. *pan, tilt, zoom* – optional floating-point parameters.

### Response syntax

The requested motion type – one of three possible values, `absolute`, `relative` or `continuous`, – should be specified in the URL path. The type should match one of supported motion types, as reported by the get node description request, see section 3.13.1.

The optional floating-point parameters `pan`, `tilt` and `zoom` define the position, translation or velocity for the respective motion types, of the requested movement. All three of that parameters may be specified. If the `zoom` is not specified, only pan and tilt movement is performed (keeping the current zoom). If the pan or tilt parameter is omitted, only the zoom change is performed (keeping the current pan and tilt position). *In other words, pan and tilt parameters should be given together. It it necessary to specify both of them to change the pan and/or tilt position of the PTZ device.*

The values of the `pan`, `tilt` and `zoom` parameters should be within the range for respective motion type and for respective motion axis, as reported by the get node descrption request described in section 3.13.1.

The call returns HTTP code 200 and an empty JSON value (`[]`) in the response body on success, or error code 500 with error text in the response body on failure.

### Response example

Move the PTZ device into an absolute position, specifying pan, tilt and zoom coordinates:

```
$ curl -X POST  "http://localhost:8880/v1/svc/cam3/ptz/move/absolute?\
pan=0.345&tilt=0.333&zoom=0"
[]
```

(Here and below the backslash denotes the line break in a UNIX command).

Increase the zoom of the PTZ camera by 10 percents, keeping the current pan and tilt position:

```
$ curl -X POST  'http://localhost:8880/v1/svc/cam3/ptz/move/relative?zoom=0.1'
[]
```

Change the pan of the PTZ camera by 15 percents, counterclockwise (note the mandatory complimentary `tilt=0`):

```
$ curl -X POST  "http://localhost:8880/v1/svc/cam3/ptz/move/relative?\
pan=-0.1&tilt=0"
[]
```

Start decreasing the tilt of the PTZ camera, slowly, at 1/5 of the maximum possible speed:

```
$ curl -X POST "http://localhost:8880/v1/svc/cam3/ptz/move/continuous?\
pan=0&tilt=-0.2"
[]
```

### 3.13.11 Stop the PTZ motion

#### Request purpose

Stop the current preset tour or previously requested continuous motion.

#### Request URL and applicable methods

`POST http://server:port/v1/svc/onvifN/ptz/stop`

#### Request parameters

none

#### Response syntax

The call returns HTTP code 200 and an empty JSON value (`[]`) in the response body on success, or error code 500 with error text in the response body on failure.

#### Response example

```
$ curl -X POST  'http://localhost:8880/v1/svc/cam3/ptz/stop'
[]
```

Here, the motion performed by ONVIF camera `cam3` is successfully stopped.

## 3.14 WebRTC signaling

The WebRTC server, whose configuration is described in section 2.3.2, exposes a few endpoints for HTTP remote calls. These include the calls for creation of a new WebRTC session (and getting so-called *SDP offer* for it, providing a newly created session with an *SDP answer* from a remote peer, and for dropping a session. There is also a call for getting a general information on the WebRTC server object.

### 3.14.1 Obtain a general information on WebRTC server

#### Request purpose

Get the information on the video sources linked with the specified WebRTC server object, and the current number of active sessions (peer connections).

#### Request URL and applicable methods

`GET http://server:port/v1/svc/`*`webrtcN`*

#### Request parameters

none

#### Response syntax

The call returns HTTP code 200 and a JSON object containing two members: `session`, containing an integer number of currently active sessions, and `live`, containing an array of strings – identifiers of live video sources linked to this instance of WebRTC server object.

#### Response example

```
$ curl -X GET 'http://localhost:8880/v1/svc/webrtc0'
{"sessions":2,"live":["cam1","cam2","cam3"]}
```

Here, the object `webrtc0` reports of 2 currently active peer connections. There are 3 live sources associated with that `webrtc0` object, with identifiers `cam1, cam2, cam3`.

### 3.14.2 Create a new session

#### Request purpose

Create a new WebRTC session. In jargon specific to WebRTC and applications running in browsers that would rougly correspond to a *peer connection*.

#### Request URL and applicable methods

`PUT http://server:port/v1/svc/webrtcN/`*`sessionID`*

#### Request parameters

*sessionID* – a random string, like UUID, identifying the session, generated by client.

#### Response syntax

The body of this request should be a JSON document of format described in section 3.14.3.

The call returns HTTP code 200 and a body of MIME type “application/sdp”, containing the *SDP offer*.

**Response example**

The next CURL command creates a new session in viinex 3.4 WebRTC server with identifier `webrtc0`. The new session receives identifier `d21a364e-33a0-4f00-9f48-d7c2bccac4b9`. The live video source requested in the new session is `cam1`.

```
$ curl -X PUT 'http://localhost:8880/v1/svc/webrtc0/\
d21a364e-33a0-4f00-9f48-d7c2bccac4b9' \
--data '{ "command": "play", "source": "cam1" }}'
```

Upon success, the viinex 3.4 responds with HTTP code 200 and the response body of content type `application/sdp` similar to the following:

```
v=0
o=viinex 1550410445215 1550410445215 IN IP4 127.0.0.1
s=-
t=0 0
a=msid-semantic: WMS d21a364e-33a0-4f00-9f48-d7c2bccac4b9
a=range:npt=now-
a=ice-ufrag:3c26
a=ice-pwd:N1SmgmEfUdjwtu2PMcKX/lXq
a=fingerprint:sha-256 22:48:0A:...:22:2F:B9:47:14
a=setup:actpass
a=candidate:a72d0b43 1 UDP 2113929471 192.168.0.70 52400 typ host
a=candidate:ba5ea35a 1 UDP 2113929471 192.168.72.1 52400 typ host
a=candidate:e1df248f 1 UDP 2113929471 192.168.120.1 52400 typ host
a=candidate:9fa9ba78 1 UDP 1677721855 148.251.0.248 52400 typ srflx
m=video 50181 UDP/TLS/RTP/SAVPF 96
c=IN IP4 0.0.0.0
a=rtcp-mux
a=sendonly
a=rtpmap:96 H264/90000
a=fmtp:96 packetization-mode=1;profile-level-id=42e01f
a=ssrc:1 msid:d21a364e-33a0-4f00-9f48-d7c2bccac4b9 cam1
a=ssrc:1 mslabel:d21a364e-33a0-4f00-9f48-d7c2bccac4b9
a=ssrc:1 label:cam1
```

This SDP document represents an *offer* from viinex 3.4 WebRTC server to the remote client (peer). The client, if it is an application running in the browser, is not required to analyze this SDP document in any specific way. Rather, the SDP offer should be passed without any modifications into the `RTCPeerConnection.setRemoteDescription()` call.

If the request body is an empty JSON object (`{}`), the WebRTC session is created, but no video stream is published in it. The RTC connection is automatically kept alive by means of keepalive STUN packets between Viinex server and a client, but this is the only traffic that is passing between them in such session. In order to publish some video stream in an empty RTC session, the API call described in 3.14.5 should be used.

Note that the sessions created by that call are in the state which requires peer to provide an answer in a timely manner. If it fails to do so within 10 seconds, the newly created session is considered expired, and resources assiciated with it are freed.

### 3.14.3 Media data request format

This section describes the format of HTTP request body for those requests to WebRTC server which have the purpose to create a session (see section 3.14.2) or change the data source for an existing session (see section 3.14.5).

Video source request for a WebRTC session should be a JSON object with the content as follows:

```
{ "command": "play" | "stop"
, "cookie": INT
, "source": STRING
          | { "name": STRING }
          | {}
, "range": "now-" | [TIMESTAMP, TIMESTAMP] | [TIMESTAMP]
, "speed": NUMBER }
```

The property `command` instructs the WebRTC session to either begin streaming video data to peer (value `"play"`), or stop streaming the data (value `"stop"`), freeing the resouces accociated with subscription to video source that might have been translated previously, but preserving the WebRTC session itself.

The property `source` can be either a string value, for using a viinex 3.4 object with respective name as a video source, or a JSON object with property `name` and its value holding the name of viinex 3.4 object, or an empty JSON object. The latter option may be used to make the WebRTC session disconnect from a video source that is currently being restreamed.

The property `range` may take either the pre-defined string value of `"now-"`, which means that the video source should be used to obtain live stream, or a pair (encoded as JSON array of 2 elements) of timestamps. The value of `range` property in form of pair of timestamps is interpreted as the time interval. For video sources assiciated with a video archive (in particular, these are video sources in a third-party VMS, see section 2.1.5 for more details), viinex 3.4 would create a connection to video archive within that third-party VMS instance and try to obtain a video stream from that archive for specified time interval. The property `range` may also take the value of array of lenth 1, which indicates the playback from the timestamp specified as the only element of the array – to the future. On other words, the syntax `range: [begin]` is equivalent to specifying `range: [begin, future]`, where `future` is some point in a very distant future.

For archive video sources, the property `speed` may also be specified to indicate the desired playback speed.

Besides the above properties, there may be specified an additional integer value `cookie`. This value is intended to identify the status of WebRTC session. Indeed, it may take some time for the session to apply changes requested via HTTP call described in section 3.14.5. An application may add a unique cookie value to such requests, and then examine the status of WebRTC session with another HTTP call described in 3.14.6. The status would contain a `cookie` value from the most recent WebRTC session change request completed by viinex 3.4 server.

There is also another syntax for this request, which was introduced initially, when WebRTC-related functionality was at first implemented in viinex 3.4. This syntax assumes that session create or update request has the form of

```
{"live": STRING}
```

or

```
{}
```

(an empty JSON object). These two forms are equivalent to the session update request of

```
{
    "command": "play",
    "source": STRING,
    "range":"now-"
}
```

and

```
{
    "command": "stop",
    "source": {}
}
```

respectively, according to the modern syntax described above. The syntax of `{"live":STRING}|{}` is considered deprecated, it may be removed from viinex 3.4 API in the future.

### 3.14.4 Provide an SDP answer for a session

#### Request purpose

Provide the WebRTC server with the SDP answer information for a specified session (peer connection).

#### Request URL and applicable methods

`POST http://server:port/v1/svc/`*`webrtcN/sessionID`*`/answer`

#### Request parameters

*webrtcN* should be the identifier of the WebRTC server object instance. *sessionID* should be the identifier for the session specified by client upon its (session) creation.

#### Response syntax

The body of this request should contain an SDP document describing the "answer" of a remote peer. It is required that said SDP answer conains information in ICE candidates generated by client for itself.

It is required that the client sets the `Content-Type` header for its request payload to the value `application/sdp`.

Upon success, the call returns HTTP status 200 and an empty JSON object. Meanwhile, having the SDP answer from the peer, the WebRTC starts the ICE connection establishment procedures, DTLS handshake and media data sending.

### Remarks

Generally, the client application, if it is an application running in the browser, is not expected to be constructing or manipulating the SDP answer to produce it to viinex 3.4 WebRTC server. Instead, the client application would use the `RTCPeerConnection.createAnswer()` call which returns a promise of the SDP data. Usually after that the `RTCPeerConnection.selLocalDescription()`. After both the remote and local descriptions are set, the browser starts searching for ICE candidates. As new candidates are found, the local session description is automatically updated. Currently the WebRTC implementation in viinex 3.4 does not support dynamical update of the list of candidates from the remote peer. It is recommended that the client application waits for all candidates are gathered, which is indicated by event `RTCPeerConnection.onicecandidate(e)` with `e.candidate==null`, and after such event has arrived – captures the `RTCPeerConnection.localDescription.sdp`, which represents the peer's SDP answer, and sends this data into viinex 3.4 WebRTC server (by means of the HTTP call being described in this paragraph).

### Response example

```
$ curl -X POST 'http://localhost:8880/v1/svc/webrtc0/
d21a364e-33a0-4f00-9f48-d7c2bccac4b9/answer' \
-H 'Content-Type: application/sdp' --data 'v=0
o=- 6542274758888401078 2 IN IP4 127.0.0.1
s=-
t=0 0
a=msid-semantic: WMS
m=video 59204 UDP/TLS/RTP/SAVPF 96
c=IN IP4 148.251.0.248
a=rtcp:9 IN IP4 0.0.0.0
a=candidate:3661447420 1 udp 2113937151 192.168.0.70 59204 typ host
generation 0 network-cost 999
a=candidate:842163049 1 udp 1677729535 148.251.0.248 59204 typ srflx
raddr 192.168.0.70 rport 59204 generation 0 network-cost 999
a=ice-ufrag:3vtT
a=ice-pwd:kwX8zta+EYN4CFdK+Z7piUry
a=ice-options:trickle
a=fingerprint:sha-256 4E:09:5A:DE:72:92:ED:FB:64:1A:01:EC:80:09:2C:A5:
C4:73:95:84:C6:0D:A7:74:C8:36:1E:99:08:5F:B3:5F
a=setup:active
a=mid:0
a=recvonly
a=rtcp-mux
a=rtpmap:96 H264/90000
a=fmtp:96 level-asymmetry-allowed=1;packetization-mode=1;
profile-level-id=42e01f
'
[]
```

Here, the WebRTC session `d21a364e-33a0-4f00-9f48-d7c2bccac4b9` previously created at object `webrtc0` is provided with an SDP answer. The answer does not contain any information on media sent by client informs the server that the peer is ready to accept the video stream using the SRTP over UDP and DTLS. The answer also contains two ICE candidates and an information on how the DTLS hanshake is going to be performed (that the remote peer is going to take client TLS role), the fingerprint of the certificate that is to be used by client.

For more information see [22].

As it is stated above, the client application needs not go into the details of the SDP document, contstruct it or manipulate it in any way. It rather should take the value of property `RTCPeerConnection.localDescription.sdp` (at the appropriate moment), and pass its value to viinex 3.4 WebRTC server.

### 3.14.5 Update an existing session

#### Request purpose

Make an update on what video stream, if any, should be streamed in specified WebRTC session.

#### Request URL and applicable methods

`POST http://server:port/v1/svc/webrtcN/`*`sessionID`*

#### Request parameters

*sessionID* – a string identifying the session which should be updated.

#### Response syntax

The body of this request should be a JSON document matching the syntax described in section 3.14.3.

The syntax and semantics of the request body matches those for HTTP call described in section 3.14.2. A body with some live video source specified in it switches the video stream within specified session to the selected one. An empty JSON object indicates that a video streaming within specified WebRTC session should be temporarily shut down, until the next HTTP call 3.14.5 is issued, or the WebRTC session is destroyed.

Upon success, the call returns HTTP code 200 and a body containing an empty JSON object or array.

The feature of switching among video streams within an existing WebRTC session provides applications with capability to build user interfaces with layouts of viewports which can be promptly switched to display various video sources upon user's request.

### 3.14.6 Get session status

#### Request purpose

Obtain the current status information on an existing WebRTC session.

#### Request URL and applicable methods

`POST http://server:port/v1/svc/webrtcN/`*`sessionID`*

#### Request parameters

*sessionID* – a string identifying the session which status should returned. The syntax of returned WebRTC session status is as follows:

```
{
   "status": "playing" | "stopped",
   "cookie": JSON,
   "last_frame": TIMESTAMP,
   "last_frame_rtp": NUMBER
}
```

The `status` property indicates whether the WebRTC session is streaming media to its peer (`"playing"`), or it is just open connection but no data is being streamed (`"stopped"`).

The `last_frame` property may hold the timestamp of the last frame sent by viinex 3.4 server to WebRTC peer within selected session. This value may be used to indicate "current position" at the timeline for video archive playback. Note that this value is reset when the WebRTC session receives a new `"play"` command, so it is guaranteed that the `last_frame` property holds either the value of timestamp for one of the frames from the requested time interval, or `null`, if no frames were sent to the peer yet after the latest `play` command was initiated by client. In no case the `last_frame` holds the timestamp from a frame sequence that was played previously, before the one which is currently being played.

Dual to the `last_frame`, property `last_frame_rtp` holds the RTP "timestamp" corresponding to the last sent video frame. RTP "timestamp" is a 32-bit unsigned integer, which can be used to measure the temporal distance between media samples sent over RTP protocol. The importance of this parameter is clear from the fact that client has no way to find out the absolute timestamp of a received frame apart from this API call, – but it's infeasible to make an HTTP call for every video frame. At the same time, there is an API at client (browser) side which allows client application to find out the RTP timestamp of a most recently displayed frame, see [29] for more information. With the `rtpTimestamp` property value known to client application, it is possible to compute the absolute timestamp of most recently displayed frame, which should be equal to

$$t_{abs} = t^0_{abs} + \frac{\tau_{rtp} - \tau^0_{rtp}}{90000\ s^{-1}}.$$

In the above equation, $t_{abs}$ is the absolute timestamp of most recently displayed frame, $t^0_{abs}$ is an absolute timestamp of some frame sent by viinex 3.4, – this value is received over this HTTP API call, and has the property name of `last_frame`; $\tau^0_{rtp}$ is the RTP "timestamp"

value corresponsing to the same frame for which the $t^0_{abs}$ was obtained, received over the same HTTP API call, and has the property name of `last_frame_rtp`; the value $\tau_{rtp}$ is the RTP "timestamp" of the most recently displayed frame, provided to client application over API described in [29]. The denominator of 90000 Hz is a constant clock rate specified in RFCs describing how the H.264 and H.265 video data should be sent over RTP.

The value `cookie` may hold the user-defined integer number. Its purpose is to identify the status of the WebRTC session and to match the status with recently commands (WebRTC session update requests) issued by client application. For more information on `cookie` field see section 3.14.3.

### 3.14.7 Gracefully shutdown a WebRTC session

#### Request purpose

Shutdown an active WebRTC session and free all resources associated with it.

#### Request URL and applicable methods

`DELETE http://server:port/v1/svc/`*`webrtcN/sessionID`*

#### Request parameters

*webrtcN* should be the identifier of the WebRTC server object instance. *sessionID* should be the identifier for the session specified by client upon its (session) creation.

#### Response syntax

Upon success, the call returns HTTP status 200 and an empty JSON object.

#### Remarks

Strictly speaking, graceful shutdown of a session is not necessary for the server to perform without resource leak. An existing peer is considered disconnected if an instance of viinex 3.4 WebRTC server fails to get a STUN keepalive response packet from that peer 10 times in a row. Such packets are sent every 1 second, – so a silently disconnected peer's session is disposed in 10 seconds after the peer disconnects. However, during that time the media data is still sent to the peer which may negatively affect the network, especially if the peer stops watching the video but remains in the same web application on the same network – there is a chance that such peer will be still receiveing media traffic which he no longer needs. This is why gracefully shutting down the WebRTC sessions should be considered a recommended practice.

#### Response example

```
$ curl -X DELETE 'http://localhost:8880/v1/svc/webrtc0/
d21a364e-33a0-4f00-9f48-d7c2bccac4b9'
```

Here, the WebRTC session `d21a364e-33a0-4f00-9f48-d7c2bccac4b9` previously created at object `webrtc0` is shut down.

## 3.15 Event storage

In order to access the events written to the database, two HTTP API call is provided for objects which implement the endpoint of type `2.9.14 EventArchive`. That API calls enable the web applications to retreive the events using a simple queries, filtering events by topics and origins, and specifying the time interval of iterset. No other means for accessing the database from HTTP API are provided. Event log cannot be changed from an API; neither events can be deleted from the log, nor there are means to access other tables besides the one which stores viinex 3.4 builtin events. Accessing other relations in the DBMS should be made by using viinex 3.4 builtin scripts. It's a deliberate decision that no means for event log modification are provided (neither via HTTP API, nor via scripting). It is possible, however for viinex 3.4 scripts to generate immutable events, which can later be retreived and interpreted by the application as a sequence of state-changing instructions. This approach can be used, for example, in alarm processing applications.

### 3.15.1 Get the summary for events stored in the database

#### Request purpose

Get the brief summary on the number of events, their origins and their topics, being stored in the database.

#### Request URL and applicable methods

`GET http://server:port/v1/svc/`*`pgN/events/summary`*

#### Request parameters

`pgN` should be the identifier of the `posgres` object instance.

Optional parameters `topic, origin, begin, end` may be used to get the summary on specific time interval, as well as on events of specific origins and topics.

#### Response syntax

Optional query parameters `topic` and `origin` should both be a comma-separated lists of strings representing event topics and event origin names, respectively. If no `topic` or `origin` parameters are specified, it is assumed that no filtering should be done on respective criteria, and the summary for events on all topics and/or origins should be retreived.

Optional query parameters `begin` and `end` may specify the temporal window for which the summary should be obtained. If any of these two parameters is omitted, it is assumed that UNIX epoch should be used as `begin` or an infinite future should be used as `end`.

The result of this HTTP request is a JSON array of viinex 3.4 objects each containing three properties: `origin`, `topic` and `count`. The meaning of these values is that the `pgN` database object contains, in a given temporal window, if it was specified in request by `begin` and/or `end` parameters, the `count` number of events of a given topic, coming from an origin of a given name.

#### Response example

```
$ curl 'http://localhost:8880/v1/svc/pg0/events/summary?begin=2021-03-27T00:00:00Z'
[{"origin":"cam1","count":1,"topic":"CounterAggregation"}
,{"origin":"cam2","count":1,"topic":"LineCrossed"}
,{"origin":"cam2","count":2,"topic":"MotionAlarm"}]
```

The `pg0` object reports that it has total 4 events recorded since midnight UTC on March 27th 2021, – with details specified for each event topic and each event origin.

### 3.15.2 Retreive events from the database

#### Request purpose

Perform a query on the database table that stores viinex 3.4 events and return results in form of JSON array.

#### Request URL and applicable methods

`GET http://server:port/v1/svc/`*`pgN/events`*

#### Request parameters

`pgN` should be the identifier of the instance of an object implementing the `2.9.14 EventArchive` endpoint.

Optional parameters `topic, origin, begin, end, limit and offset` may be used to filter events that need to be retreived by their topic, origin, timestamp, as well as perform pagination in case if there too many events to be returned in a single query.

#### Response syntax

Optional query parameters `topic` and `origin` should both be a comma-separated lists of strings representing event topics and event origin names, respectively. If no `topic` or `origin` parameters are specified, it is assumed that no filtering should be done on respective criteria.

Optional query parameters `begin` and `end` may specify the temporal window from which the events should be retreived. If any of these two parameters is omitted, it is assumed that UNIX epoch should be used as `begin` or an infinite future should be used as `end`.

Optional parameter `order` may take one of two values, `asc` or `desc`, and serves for the purpose of specifying how the events should be sorted when being retreived from the database.

The events are always sorted according to their timestamp. The value of `order=asc` explicitly specifies that events should be retreived in timestamp ascending order, while parameter `order=desc` specifies that events should be retreived in timestamp descending order. When parameter `order` is omitted, the ascending sort order is assumed.

Optional parameters `limit` and `offset` allow to perform a pagination when querying for a large volume of events. The `limit` parameter specifies the maximum number of events to be retreived from the database in one single query. The `offset` parameter specifies how many events need to be skipped, taking into account other query parameters and the sort order, before actually retreiving and returning requested events. When no `limit` parameter is specified, it is assumed that the number of events to be returned is limited to 1000.

The result of this HTTP request is a JSON array of viinex 3.4 event objects. Each event object has mandatory properties `timestamp`, `topic`, `origin` and `data`. In more detail this syntax is described in section 3.18.

### Remarks

When the `origin` parameter is specified, this filter is applied only to the `origin.name` property of events.

No means are provided to filter events based on their `data`. Therefore, such filtering should be performed at client side.

Since the results are returned in a single query, there are no measures taken at the server side in order to retrieve results from database page-wise, or to deal with cursors, and so on. The query is being executed as is, and all the results are fetched from the database, formatted into JSON and sent to the client.

This means that an application should take additional care of the number of events fetched and returned in a single request. The implicit value of `limit` is deliberately chosen to be small. An application may explicitly establish a large `limit` to retreive a large number of events in a single query, though this is not recommended. Instead, it is advised that pagination is used for accessing potentially large set of events. The query returning a large number of events may result in resouce exhaustion and performance degradation or denial of service at viinex 3.4 server side.

### Response example

```
$ curl -X GET 'http://192.168.0.71:8880/v1/svc/pg0/events?
origin=cam1,cam2&topic=RtspException,MotionAlarm
&begin=2021-03-17T00:00:00Z&limit=100&offset=700'
[{"origin":{"name":"cam1",
            "details":{"Source":"VideoSource_1"},
            "type":"onvif"},
  "data":{"state":false},"topic":"MotionAlarm","timestamp":"2021-03-17T06:37:10Z"},
 {"origin":{"name":"cam1",
            "details":{"VideoSourceConfigurationToken":"VideoSourceToken",
                       "Rule":"MyMotionDetectorRule",
                       "VideoAnalyticsConfigurationToken":"VideoAnalyticsToken"},
            "type":"onvif"},
  "data":{"state":false},"topic":"MotionAlarm","timestamp":"2021-03-17T06:37:10Z"},
 {"origin":{"name":"cam1","details":{"Source":"VideoSource_1"},"type":"onvif"},
```

```
  "data":{"state":false},"topic":"MotionAlarm","timestamp":"2021-03-17T06:37:11Z"},
 ...
 {"origin":{"name":"cam1",
            "details":{"VideoSourceConfigurationToken":"VideoSourceToken",
                       "Rule":"MyMotionDetectorRule",
                       "VideoAnalyticsConfigurationToken":"VideoAnalyticsToken"},
            "type":"onvif"},
  "data":{"state":true},"topic":"MotionAlarm","timestamp":"2021-03-17T06:42:38Z"}]
]
```

The query to extract the 7th 100-events-length page, starting from midnight UTC on March 17th 2021, timestamp ascending order) from sources `cam1` and `cam2` of topics `MotionAlarm` and `RtspException` is executed, and respective events are returned as the result of HTTP request.

## 3.16 Authentication details and ACL-related storage

This section describes the API which can be implemented by database integration objects described in section 2.5.2. This API is divided into two parts represented by endpoints 2.9.2 `AclProvider` (for reading out the ACL-related data, that's first five calls described in this section) and 2.9.3 `AclStorage` (for modifying that data, that's last two calls described in this section). Such division is implemented deliberately in order to make it possible to separate the privileges of reading the access control-related information and for modifying it.

The ACL-related information stored within database can be classified as credentials database, users-to-roles mapping, and the access control list. It should be noted that for practical reasons viinex 3.4 allows for storing multiple access control lists within same database. An access control list is identified by an integer number, which is arbitrarily assigned by administrator. This number should be specified as the value of property `acl_id` in the configuration of authentication and authorization provider object, as described in section 2.7.

The credentials list, is common (shared) for all access control lists within same database. This lets same users have different roles in different ACLs. On the other hand, the users-to-roles mapping and the set of access control entries is separate for each ACL with a specific identifier. No specific action needs to be taken to begin using a specific new ACL identifier – these IDs need not be registered or explicitly created. From client applications perspective it may be safely assumed that an ACL with a specific ID already exists. If it does not, – viinex 3.4 would behave as this ACL exists but contains no roles and no access control entries.

For purpose of caching and conflict solving (when ACL information gets edited by users), viinex 3.4 keeps track of ACL information version number. This version number may be thought of as a revision number in a centralized version control system (like SVN). When ACL information within database is modified, a new version number is assigned to that ACL. It is required that the client to this API, when a call to modify ACL information is being made, specifies the base version of ACL information (i.e. the version which is "current" to that client, and which is actually being modified). This prevents the situation when two clients make the change to ACL without knowing of other's changes, which can render the information in ACL database inconsistent. The simple schema implemented in viinex 3.4 assumes that if a concurrent edit happened, the client attempting to edit an outdated version of ACL gets an error from server. After that this client can retreive the new ACL infromation, apply his edits and verify the consistency of ACL before writing it to database.

Only the users-to-roles mapping and the set of access control entries are subject to this version number tracking. The version of each ACL with its own identifier is tracked independently of others (i.e. changes made to one ACL in the same database do not affect version number of other ACLs). The users' credentials information is not subject to version tracking; it can be updated any time independently. The reason for this is that each user's credentials are viewed as information not dependent on other, and it's atomic, so a "last write wins" conflict resolution strategy can be safely applied for it without compromising data consistency.

### 3.16.1 Enumerate users accounts

#### Request purpose

Get the information on authentication details from the database

#### Request URL and applicable methods

`GET http://servername:port/v1/svc/`*`dbN/acl/accounts`*

#### Request parameters

none

#### Response syntax

There are no parameters to this API call. The result of this call is a JSON array of records, each holding authentication information for a specific account. The format of each record is described in section 2.7.3. Each record may represent a "human" account intended to contain a password (the type of such record would be `password`), or a "bot" account, which is also intended for password authentication, but password can be a lot longer than for a human and therefore need not be stored in hashed form (the record type for this case would be `apikey`).

In either case, for security reasons the secret part of the record (which is called `secret` for `apikey` records and `digest` for `password` records) is redacted before being sent to the caller. There are no means to find out the password or hash password via this API. Once stored in the database, this secret part is never sent back by viinex 3.4 server to any client.

#### Response example

```
$ curl http://localhost:8881/v1/svc/db0/acl/accounts
[{"secret":"redacted","key":"agent1","type":"apikey"}
,{"secret":"redacted","key":"agent2","type":"apikey"}
,{"digest":"redacted","realm":"Viinex","type":"password","login":"gzh"}]
```

### 3.16.2 Get the ACL version

#### Request purpose

Get the value identifying the version of information stored in database for ACL with specified identifier

#### Request URL and applicable methods

`GET http://servername:port/v1/svc/`*`dbN/acl/ACL_ID/version`*

#### Request parameters

`ACL_ID` – numerical identifier of the access control list – is a part of URL path

#### Response syntax

There are no query parameters to this API call, except for ACL identifier in the URL path.

The result of this call is a JSON object holding the ACL identifier and the version number for that ACL.

#### Response example

```
$ curl http://localhost:8881/v1/svc/db0/acl/1/version
{"acl_id":1,"version":6}
```

In this example, the ACL with identifier 1 is examined for version number. The returned version number is 6.

#### Remarks

The client software should not make any assumptions on how next version number is chosen. Current implementation simply increments a version number by 1 every time a change to the ACL data is made. However this algorithm may be changed any time without notice.

### 3.16.3 Get the users-to-roles mapping

#### Request purpose

Get the list of user–role associations stored in database for ACL with specified identifier

#### Request URL and applicable methods

`GET http://servername:port/v1/svc/`*`dbN/acl/ACL_ID/roles`*

#### Request parameters

`ACL_ID` – numerical identifier of the access control list – is a part of URL path

#### Response syntax

There are no query parameters to this API call, except for ACL identifier in the URL path.

The result of this call is a JSON array of pairs of strings, each encoded as a JSON array consisting of exactly 2 elements, where first element represents user identifier (login or api key), and second element represents role identifier.

#### Response example

```
$ curl http://localhost:8881/v1/svc/db0/acl/1/roles
[["agent2","user"]
,["admin","administrator"]
,["user1","user"]
,["root","administrator"]]
```

In this example the mapping for four users is returned within access control list with identifier 1. There are two active roles in that ACL, the ``user'' role, of which the members are users with logins ``user1'' and ``agent2'', and the ``administrator'' role, which is granted to users with logins ``admin'' and ``root''.

#### Remarks

The semantics of data returned from that call matches that described for section `roles` of configuration of `authnz` object, as described in section 2.7.3. However the Cartesian product syntax described there is not recognized and is never produced by API calls.

### 3.16.4 Get the access control entries

#### Request purpose

Get the list of access control entries stored in database for ACL with specified identifier

#### Request URL and applicable methods

`GET http://servername:port/v1/svc/`*`dbN/acl/ACL_ID/entries`*

#### Request parameters

`ACL_ID` – numerical identifier of the access control list – is a part of URL path

#### Response syntax

There are no query parameters to this API call, except for ACL identifier in the URL path.

The result of this call is a JSON array of records representing access control entries (ACEs). Each ACE has the syntax of a tuple (triplet), represented as a JSON array of exactly three elements, of which the first element represents the role identifier for which the ACE is specified, the second element could be a string containing viinex 3.4 object ID, to which this ACE should be applied, or value `null` if the ACE should be applied to all objects; and the tird element of the ACE tuple should be the string representing the type of endpoint, access to which is granted by this ACE, or `null` if access to endpoints of all types should be granted.

The valid strings for referring to viinex 3.4 endpoint types are enumerated in section 2.9 of this document.

#### Response example

```
$ curl http://localhost:8881/v1/svc/db0/acl/1/entries
[["administrator",null,null]
,["user",null,"VideoSource"]
,["user","cam1",null]
,["user",null,"MetaConfigStorage"]]
```

In this example four access control entries are returned for ACL with identifier 1. The first ACE permits the access to all endpoints an all objects for users to whom the role ``administrator'' was granted.

The second ACE permits the access to endpoint 2.9.38 `VideoSource` to principals who are member of role ``user''. Also, for that principals full access to object ``cam1'' is granted by the third ACE. Finally, the fourth ACE permits members of role ``user'' to access endpoint 2.9.20 `MetaConfigStorage`, which is actually important to find out via HTTP API which viinex 3.4 objects and endpoints are published within the HTTP server.

### 3.16.5 Get all ACL information in one call

#### Request purpose

Get the version of ACL data, the user-roles mapping and access control entries of that ACL in one HTTP call

#### Request URL and applicable methods

`GET http://servername:port/v1/svc/`*`dbN/acl/ACL_ID`*

#### Request parameters

`ACL_ID` – numerical identifier of the access control list – is a part of URL path

#### Response syntax

There are no query parameters to this API call, except for ACL identifier in the URL path.

The result of this call is a JSON object with properties `version`, `roles` and `entries` set to result of methods 3.16.2, 3.16.3 and 3.16.4 respectively.

This API method is only provided to save the number of roundtrips. It can be used if the ACL database is small so the overhead for receiving and parsing the data is negligible. Otherwise it might be more preferrable for clients to use methods returning each type of information separately.

#### Response example

```
$ curl http://localhost:8881/v1/svc/db0/acl/1
{
  "version": 6,
  "roles":
    [["agent2","user"]
    ,["admin","administrator"]
    ,["user1","user"]
    ,["root","administrator"]],
  "entries":
    [["administrator",null,null]
    ,["user",null,"VideoSource"]
    ,["user","cam1",null]
    ,["user",null,"MetaConfigStorage"]]
}
```

Note that, as mentioned in the beginning of this section, credentials information is not considered a part of ACL, not associated with an ACL ID, and not returned by this method.

### 3.16.6 Modify users accounts

#### Request purpose

Get the information on authentication details from the database

#### Request URL and applicable methods

`POST http://servername:port/v1/svc/`*`dbN/acl/accounts`*

### Request parameters

none

### Response syntax

There are no parameters to this API call within the request URL or headers.

The body of the request should be a JSON array holding one or more records of form

```
["add" | "remove", AuthDetails, ...]
```

Each record is an array where the first element is a string representing the action which needs to be taken, and second and all subsequent elements form a list of subjects for which the action should be taken. The format of `AuthDetails` record is described in section 2.7.3 and matches the record format being output by the call 3.16.1, except that the secret parts of the records are expected to be meaningful (not redacted) in this call when the action verb (the first element of array is equal to `add`.

As the result of this call, the authentication details listed with `"add"` action get added or updated in the credentials database, and the authentication details records listed with the `"remove"` action – get removed from credentials database.

Note that credentials table within the database has the uniqueness constraint set for the (user id, realm) pair. That is, there can be more than one record for the same user id, as long as realm is different. (The records of type `apikey` a considered as records with realm of empty string within that table). Respectively, the INSERT/UPDATE and DELETE operators are executed on the table so that this (user id, realm) pair is considered a primary key.

When removing authentication detail records, the secret part of records passed to this call is ignored, that is – the password or is digest is not checked for equality with what caller has specified when the record gets removed.

The result of this call is a JSON object of the form

```
{"modified": INT}.
```

The numeric value returned in that manner reports how many records in database were modified while executing this call.

### Response example

```
$ curl -X POST http://localhost:8881/v1/svc/db0/acl/accounts \
--data-binary '[
  ["add",
    {
      "type":"password",
      "login":"user1",
      "realm":"viinex",
      "digest":"ebb55e64ddba90a850bf863a18e79aaa"
    },
    {
```

```
      "type":"password",
      "login":"user2",
      "realm":"viinex",
      "digest":"67a8bec28d39ba9cbb355c331fce07c0"
    }
  ],
  ["remove",
    {
      "type":"apikey",
      "key":"agent1",
      "secret":""
    }
  ]
]'
{"modified":3}
```

This example contains instructions for credentials database to add two account records for `user1` and `user2`, with specified realm ``viinex'' and password digests, and to remove bot account `agent1`.

#### Remarks

Note that for authentication details records specified for removal, the field `digest` or `secret` needs to be specified, for the server side to be able to parse records of that type. The content of these fields may be left empty, it is ignored for account records being removed.

### 3.16.7 Modify an access control list

#### Request purpose

Get the information on authentication details from the database

#### Request URL and applicable methods

`POST http://servername:port/v1/svc/`*`dbN/acl/ACL_ID`*

#### Request parameters

`ACL_ID` – numerical identifier of the access control list – is a part of URL path

#### Response syntax

There are no query parameters to this API call, except for ACL identifier in the URL path.

The body of the request should be a JSON object holding one mandatory parameter, `version`, and two optional parameters, `roles` and `entries`:

```
{
    "version": INT,
    "roles": [["add" | "remove", [STRING,STRING], ...], ...],
    "entries": [["add" | "remove",
                   [STRING, STRING | null, STRING | null],
                   ...
                ], ...]
}
```

The `version` property should have the value returned by the call to method 3.16.2. This is required to make sure that the client modifies the current version of ACL, to prevent the storing of inconsistent data on concurrent edits.

The properties `roles` and `entries` may each contain the list of edits for user–role mapping and for the access control entries within the ACL, respectively. Each of such edits, like in method 3.16.6, is represented by an array, where first element defines an action that needs to be taken (`"add"` for adding of a user–role association or an access control entry, or `"remove"` for removal of respective element), followed by one or more elements that need to be added or removed.

Upon success, this method returns the summary on the records modified in the database, for user–role associations and for ACEs explicitly:

```
{
    "modified": {
        "roles":INT,
        "entries":INT
    }
}.
```

**Response example**

```
$ curl -X POST http://localhost:8881/v1/svc/db0/acl/1 \
--data-binary '
{
  "version":8,
  "roles":[
    ["add",
      ["user","user1"],
      ["root","administrator"]
    ],
    ["remove",
      ["gzh","user"]
    ]
  ],
  "entries":[
    ["add",
      ["user","cam2",null],
      ["user",null,"SnapshotSource"]
    ],
    ["remove",["user",null,null]]
  ]
```

```
}'
{"modified":{"roles":0,"entries":2}}
```

In this example, a client assumes that the current version of data for ACL with identifier “1” is 8. The client instructs the server to add two user–role associations, and remove one. Also two access control entries are added (for the role ``user'' to have full control over object ``cam2'', and also to access endpoints of type 2.9.33 SnapshotSource on all objects. One ACE is also removed from database (the one which would, if present, grant all access on all objects to members of role ``user'').

#### Remarks

Besides the ACL information version tracking, no consistency checks are performed by server when modifying the data. This means that the server does not check role names, user names and object names. It's also up to client to validate the permissions assigned by each access control entry, and make sure these permissions do not overlap (or overlap, if this is required). Since ACE overlapping is considered legitimate, – viinex 3.4 does not attempt to detect duplicate records in access control entry table.

## 3.17 Abstract interfaces

This section describes two simple API calls – read and update – which are not bound to a specific object implementation, but are used in some simple cases by viinex 3.4 built-in objects, and, most notably, by scripts. Each script represents a state machine and can publish a part (a view) of its state to be read by external software, and it can also accept the synchronous requests via HTTP which can be interpreted as requests for state change (update) and possibly for some other actions available from the scripting engine.

### 3.17.1 Stateful

#### Request purpose

Obtain the state of an object

#### Request URL and applicable methods

`GET http://SERVER:PORT/v1/svc/`*`objectN`*

#### Request parameters

none

#### Response syntax

The above call gives a read access to the state information of an object, which might be published by that object. The syntax of response body of this HTTP call is always JSON, however the exact form of the response depends on the object implementation.

In particular, the `script` object provides the `Stateful` interface implementation in viinex 3.4. Each script defines which data should be published as its "state" – it does so in the source code of the script, by means of a call to the function `vnx.publish()`.

Note that once the script publishes its "state", this HTTP request call can be served to as many clients as needed, in parallel, – an application does not need to worry about the performance of this request, because the published state is copied and stored by viinex 3.4 as immutable data, till the next JavaScript call to `vnx.publish()`.

### 3.17.2 Updateable

#### Request purpose

Update the state of an object, and possibly perform other actions (achieve required side effects).

#### Request URL and applicable methods

`POST http://SERVER:PORT/v1/svc/`*`objectN`*

#### Request parameters

none

#### Response syntax

The request body for this call is always JSON, but the form of JSON data that needs to be passed as request body is defined by each object implementation individually. In particular, the scripts would typically receive this update as a call to the function `onupdate()`. The JSON data received as the HTTP body of this request is passed as an argument to that JavaScript call. It is up to the implementation which JSON data structure is considered valid.

The response to this call is also always a JSON, but its semantics depends on the implementation. For scripts, the return value from the `onupdate()` function is sent as the HTTP body in response to this request.

Note that, particularly for the scripts, which are essentially single-threaded, since the script state machine update requires executing some part of script code, – the HTTP calls to the `Updateable` interface of an object are serialized. In other words, if there are concurrent clients' HTTP update calls to the same `Updateable` object (for example, an instance of a script) – these concurrent calls will be processed sequentially, one after another. This differs from how the HTTP requests for reading out the state are processed (see section 3.17.1).

## 3.18 WebSocket interface

In order to acquire real-time events from viinex 3.4, the latter implements additional application-level interface on top of WebSocket protocol.

In order to use the WebSocket interface implemented by viinex 3.4, a client application should establish the WebSocket connection to that server. That connection should be established to the URL `http://SERVER:PORT/` in order to work with the objects created in static configuration of viinex 3.4 instance and published by the webserver listening on the port number `PORT`. Otherwise, in order to work with the objects created dynamically in a cluster with name *CLUSTER_NAME*, the client should make a WebSocket connection to the endpoint `http://SERVER:PORT/v1/cluster/`*CLUSTER_NAME*.

The application interface implemented by viinex 3.4 is straightforward. The client application can issue commands to alter its state, and can receive events from viinex 3.4 server. The interchange is performed in form of WebSocket data frames of type `text`, see [15]. The syntax of the text data sent by viinex 3.4 server and expected by such server from client is JSON values, with semantics described below. The interchange completely asynchronous: the server never sends anything as a "response" to client's request.

The client's WebSocket requests recognized by viinex 3.4 server should have the form of JSON array, whose first element should be a string defining an action, and the rest elements are interpreted as the arguments for corresponding action:

```
["ACTION", PARAM_1, ...]
```

There is also an exception from this rule – an empty array, `[]`, which can be sent by client as a heartbeat. It is recommended that viinex 3.4 client sends the heartbeat message to the server every 20 seconds. If viinex 3.4 cannot recognize client's request acquired via WebSocket protocol, it closes the connection to that client.

Valid client's requests are: `disconnect`, `authenticate`, and `subscribe`.

The `disconnect` request instructs the server to gracefully close the connection. It foes not require additional arguments.

The `authenticate` request is required if viinex 3.4 web server is configured to require authentication. With this request, the client should pass the value of `auth` cookie set by viinex 3.4 web server in the reponse to authentication request, as described in section 3.2.2. This should be a string containing a base64-encoded JSON structure (the authentication response). As an alternative, the client may set the argument of `authenticate` WebSocket request to the authentication response itself, also returned by viinex 3.4 server as the body of HTTP request described in 3.2.2. So, the `authenticate` request could look like

```
["authenticate", "eyJzYWx0IjoiMzQxNDcwMzE5ZjVlZDRhODIxM2UzMjdh\
MWUyNTJiODJlYzhhZmFmZGM4MWYzMzQxNTNjZmE5YzU5YmFlMmNhMSIsImlzc3\
VlZCI6IjIwMTctMDEtMTZUMTI6NTg6MzkuNzcyODI0NFoiLCJzaWduIjoiYWMz\
MmZkYTc1NDViZDlhNGExMTQ5YzRjOWVjOTkzNWEiLCJ1c2VyIjoxfQ=="]
```

or, an equivalent form,

```
["authenticate", {
  "salt":"341470319f5ed4a8213e327a1e252b82ec8afafdc81f334153cfa9c59bae2ca1",
```

```
  "issued":"2017-01-16T12:58:39.7728244Z",
  "sign":"ac32fda7545bd9a4a1149c4c9ec9935a",
  "user":1}]
```

If the server requires authentication, this request should be issued first after the client connects via WebSocket protocol. If viinex 3.4 server cannot validate authentication cookie, it terminates the connection.

The `subscribe` request is used by the client to establish or change its subscription to events from viinex 3.4 server. This request may be accompanied with an optional argument defining the server-side filter for events. If omitted, it is assumed that all events are transmitted to the client.

The filter of a subscription is defined by JSON object with two optional values, `origins` and `topics`. The `origins` value may be set to a JSON array holding the list of viinex 3.4 object names (identifiers), sourcing the events to be received by the client. If this value is absent, it is assumed that the client should receive events from all origins. The `topics` value may be set to a JSON array holding the list of event topics in which the client is interested. The topics of events match that described in section 2.4.3. If this parameter is absent, it is assumed that the client is interested in events of all topics. Here are some examples for the `subscribe` requests:

```
["subscribe", {}]
```

— receive all events from all objects;

```
["subscribe", {"origins": ["cam1","cam2"]}]
```

— receive all events originating from viinex 3.4 objects with names `"cam1"` and `"cam2"` (which could be ONVIF cameras or raw video sources);

```
["subscribe", {"origins": ["cam1"],
               "topics": ["MotionAlarm", "GlobalSceneChange"]}]
```

— receive the events from motion detector and global scene change detector working for object `"cam1"`.

In order to begin receiving some events, it is necessary that client issues the `subscribe` command after it is connected (and authenticated, if required). Initially, the new client is not subscribed to anything, and will not receive any events without explicit action.

Note that not only the client's subscription matters on what objects' events are sent to the client, but also the server configuration, in particular — which objects are linked to the instance of the web server. The events from a camera are only sent to the client via the WebSocket connection only if that connection is established to viinex 3.4 web server instance which is linked with that camera, as described in section 2.10 of this document.

After the subscription is established, the server starts to transmit the events from linked event sources to the WebSocket client. Each event is transmitted in a separate WebSocket frame of type text, holding the JSON record of the following form:

```
{
  "timestamp": TIMESTAMP,
  "origin": {
```

```
    "name": STRING,
    "type": STRING,
    ...
  },
  "topic": STRING,
  "data": OBJECT
}
```

The `timestamp` contains the time when the event was produced. The `origin` value is an object containing the string property `name` identifying the viinex 3.4 object which produced that event; there is also the string value `type`, defining the type of event's origin, and can be some more optional values depending on origin's type (for instance, in case of ONVIF events, the device may report the identifier of a specific video detector or rule which has triggered the alert). The `topic` property contains a string type of event's topic, as specified in section 2.4.3 of this document. Finally, the `data` is a JSON object whose form depends on event topic, but for simple binary-state detectors it has the form of

```
{ "state": BOOLEAN }
```

indicating whether the alert generated by specific detector is active or not.

Additionally, the server may send an empty JSON object, `{}`, as a heartbeat, if there were no events matching the client's subscription within the last 30 seconds. The client may use this as an indication of connection health.

An example of sequence of events that may be received by a client is given below:

```
{"timestamp":"2017-11-13T13:38:54.9068374Z",
 "origin":{"name":"raw0","type":"localvideo"},
 "data":{"state":true},"topic":"MotionAlarm"}
{"timestamp":"2017-11-13T13:38:56.1068374Z",
 "origin":{"name":"raw0","type":"localvideo"},
 "data":{"state":true},"topic":"MotionAlarm"}
{"timestamp":"2017-11-13T13:38:56.5068374Z",
 "origin":{"name":"raw0","type":"localvideo"},
 "data":{"state":false},"topic":"MotionAlarm"}
{}
{}
```

It is possible to acquire similar dump by means of a simple WebSocket client, for instance like the one available for Google Chrome browser: https://chrome.google.com/webstore/detail/simple-websocket-client/pfdhoblngboilpfeibdedpjgfnlcodoo

## 3.19 Configuration clusters

Configuration clusters are the way to create and dispose groups of viinex 3.4 objects possibly linked to each other and working together, but isolated from the objects created in other clusters and/or main (static) configuration of viinex 3.4 instance. That is, each cluster resemble an instance of viinex 3.4 with some custom static configuration, with the difference that it can be created and destroyed in the runtime, without restarting the instance of viinex 3.4. However,

the configuration of objects created within a cluster cannot be changed, once the cluster is created. The only way to change that configuration is to stop (destroy) the cluster and create it anew with an altered configuration.

The reason for such isolation of objects belonging to different clusters and the immutability of objects' configuration is that this simplifies the configuration semantics: there is a guarantee that objects created within the same cluster (or static main configuration) are always available to each other, always functioning, so that the links between such objects, if described in the configuration, are valid as long as objects are running.

The configuration clusters are transient, they do not survive the restart of viinex 3.4 instance and thus should be re-created explicitly every time such restart is done.

### 3.19.1 Enumerate existing clusters

#### Request purpose

Obtain the names of dynamic configuration clusters created so far

#### Request URL and applicable methods

```
GET http://servername:port/v1/cluster
```

#### Request parameters

none

#### Response syntax

JSON array of strings:

```
[NAME_1, NAME_2, ...]
```

each string representing a cluster. There is always one special name `"main"` contained in that list, which represents the main configuration (statically set via configuration files).

#### Response example

```
["main", "cluster1", "cluster3"]
```

### 3.19.2 Create a new cluster of objects

#### Request purpose

Obtain the names of dynamic configuration clusters created so far

#### Request URL and applicable methods

```
PUT http://servername:port/v1/cluster/NAME
```

#### Request parameters

NAME – the name (identifier) for the cluster to be created. HTTP request body should contain the configuration for newly created cluster

#### Response syntax

The body of this HTTP request should contain the configuration for the newly created cluster. This is a JSON document with syntax and semantics described in the first chapter of this manual. Note that the configuration should come in one single JSON document, containing all necessary `objects` and `links` at once. Split configuration cannot be processed when creating the cluster via HTTP API.

#### Response example

```
$ curl -X PUT http://localhost:8880/v1/cluster/ClusterOne \
       --data-binary @ClusterOne-config.json
```

– dynamically creates the cluster with name `ClusterOne`, taking the configuration for objects and links within that cluster from the local file `ClusterOne-config.json`. The objects and links described in that configuration file are created and started. A simple example of such configuration is given in section 2.5.6.

### 3.19.3 Remove an existing cluster of objects

#### Request purpose

Remove an existing cluster of objects, stopping and destroying all objects in that cluster and links between them.

#### Request URL and applicable methods

```
DELETE http://servername:port/v1/cluster/NAME
```

#### Request parameters

NAME – the name (identifier) for the cluster to be removed.

**Response example**

```
$ curl -X DELETE http://localhost:8880/v1/cluster/ClusterOne
```

– dynamically stops all the objects previously clreated in cluster with name `ClusterOne`, destroys all links between that objects and removes the cluster itself. A new cluster with the same name can be created after that.

### 3.19.4 Enumerate components published by a cluster

**Request purpose**

Obtain the list of components published by a cluster, along with their interface types.

**Request URL and applicable methods**

`GET http://servername:`*`port`*`/v1/cluster/`*`CLUSTER_NAME`*

**Request parameters**

none

**Remarks**

The semantics and output syntax for this request matches that for the request 3.1.1 which can be used for enumerating the components published under a specific webserver in a static configuration. The difference is that this requests provides information for the objects published in a specified dynamically created cluster.

### 3.19.5 Obtain the metainformation on components published by a cluster

**Request purpose**

Obtain the metainformation previously stored in the configuration sections for objects created and published within the specified cluster and published under this instance of web server.

**Request URL and applicable methods**

`GET http://servername:port/v1/cluster/`*`CLUSTER_NAME/meta`*

#### Request parameters

none

#### Remarks

The semantics and output syntax for this request matches that for the request 3.1.2 which can be used for enumerating the components published under a specific webserver in a static configuration. The difference is that this requests provides information for the objects published in a specified dynamically created cluster.

### 3.19.6 Access viinex 3.4 objects in configuration clusters

#### Request purpose

Forward the HTTP calls to a specific objects to that objects within a specified cluster.

#### Request URL and applicable methods

*METHOD* `http://servername:port/v1/cluster/`*CLUSTER_NAME/OBJECT_NAME/PATH?QUERY*

#### Request parameters

CLUSTER_NAME – the identifier of a cluster, OBJECT_NAME – the identifier of an object. PATH and QUERY could be an object-specific and request-specific parameters. There can also be request-specific paramters in the HTTP request body.

#### Response syntax

Any HTTP call having the URL path prefix of `/v1/cluster/`, and then a cluster identifier, followed by a slash an object identifier, with optional URL path suffix, query parameters and HTTP request body, are processed as if the object with respective identifier exited in the static (main) configuration, and the request was performed to `/v1/svc/` URL prefix, instead of `/v1/cluster/CLUSTER_NAME/`.

If the cluster with specified name does not exist, or an object with specified name does not exist in that cluster, an error with code 404 is returned.

#### Response example

```
$ curl http://localhost:8880/v1/cluster/ClusterOne/cam1/stream.m3u8
```

receives the HLS playlist for live stream of video source `cam1` created in the configuration cluster `ClusterOne`.

**Remarks**

In order for objects declared in the cluster's configuration to be published under the webserver declared in the static configuration of viinex 3.4, such objects should be linked with an object of type `publish` which should be declared in their cluster. For more information refer to section 2.5.6.

### 3.19.7 Obtaining events from a cluster

In order to receive the events from the objects published by a specified cluster with name *CLUSTER_NAME*, a client should establish a WebSocket connection to the endpoint `http://SERVER:PORT/v1/cluster/`*`CLUSTER_NAME`*. Further interaction with the viinex 3.4 WebSocket server is performed as described in section 3.18. The events from the objects created in a specific cluster are segregated from the events produced by other clusters and/or the static configuration, so no client receives events from different clusters in a single WebSocket connection.

## 3.20 Scripting-style API endpoint

The next two chapters – 4 ("Scripting and JS API") and 5 – ("WAMP interface") – are built around somewhat different approach to viinex 3.4 API syntax, compared to the HTTP API described in this chapter. The major stylistical difference is that, while HTTP API of viinex 3.4 has been developed to look terse and "RESTful", – the way of how routing is organized and how the parameters get passed into HTTP API calls may sometimes be hard to generalize.

In contrast, JS API for viinex 3.4 builtin scripts and WAMP API provided by viinex 3.4 are literally using the same internal description of viinex 3.4 objects and their programming interfaces, and thus have well defined rules for how the objects are referred to for a call to be made (see sections 4.2.7 and 5.3.2), and how parameter are passed into the calls.

This section establishes a bridge between HTTP transport and scripting-style API of viinex 3.4, making it possible to use similar approach for routing and data marshalling.

Namely, only one new HTTP endpoint is introduced, that is the `POST` method on the URI `/v1/svc` (for the main cluster) or `/v1/cluster/CLUSTERNAME` (on a dynamically created cluster).

The body of such request should have the form of array of 3-tuples, each encoded as a JSON array, where each tuple defines a call to be made on a viinex 3.4 object within respective cluster. The elements of each tuple (inner array) are:

- name (identifier) of an object to be called – a string;
- (optionally, interface/endpoint name, and) method name of the object to be called. It should be a string having the form of `"methodName"` or `EndpointName.methodName` (interface/endpoint name and method name separated with a dot). The same rules apply for method name resolution as described in section 4.2.7: endpoint name can be specified for more strictness and in case of ambiguity, but it typically can be omitted.
- call arguments, encoded as an array of JSON values. If the method does not require any

arguments, this third element of the tuple can be omitted, or, alternatively, no arguments can be encoded as an empty array.

The response body of the call is a JSON array, holding the results for each call request that was provided in the request.

In this way the single HTTP request effectively allows several methods on different viinex 3.4 objects to be invoked. This implies three important implications on the way of how the calls are handled.

First, viinex 3.4 may handle the invocation requests which come in one HTTP call concurrently, in parallel. There is no notion of sequence in such invocations; no guarantees are provided on the order of execution of methods being called. If the application needs such guarantees – it should issue respective HTTP requests sequentially, one after receiving results from another. The order of tuples (object name, method name, arguments) in the array which is sent as the request body can be arbitrary and is only used to establish the correspondence of the method calls with their results.

Another consequence is that, unfortunately, staying with HTTP/1.1 paradigm, there's no easy way to return the results of the method invocations which have been handled faster than others: viinex 3.4 has to wait for all the requested invocations complete in order to return the results for all of them. In other words, with this HTTP API endpoint, the time required to get the results for many remote method calls will be no less than the time required to complete the longest of these calls. If you need a different behavior in your app – you may want to consider using the WAMP interface, as WAMP is based on completely asynchronous model for remote method invocation: the communcation channel isn't blocked while one method is being processed, so other methods can be called.

Finally, this HTTP call itself always "succeeds" (well, almost – it requires the same permissions as the call to `GET /v1/svc`), and the errors are handled separately for every method being invoked. If an error occurs for one of the methods being invoked – it does not affect the execution of other methods, at least at the level of RPC transport. This includes the permission check errors: it's possible to request two calls on different interfaces ("endpoints") of the same object or different methods of different objects, while having permissions to do one call and lacking permissions to do the other one. In that case, the call for which permission checks are successful will be performed and its result will be returned, while the for method which can't be called – the permission error will be returned, – each on its corresponding position of the array holding the results and sent back are HTTP response.

An example of how this HTTP endpoint can be used is given below:

```
$ curl http://192.168.32.106:8880/v1/svc -X POST \
  --data-binary '[["stor_106_0","VideoStorage.summary"],
                  ["stor_106_0","channelSummary",["cam_106_3"]],
                  ["cam_106_3", "snapshotBase64"]]' | jq
[
  {
    "contexts": {
      "cam_106_3": {
        "disk_usage": 6502878483,
        "time_boundaries": [
          "2024-03-15T21:35:41.384Z",
          "2025-04-08T01:32:28.825Z"
        ],
```

```
        "timeline": null
      },
      ...
    },
    "disk_free_space": 354410430464,
    "disk_usage": 1417709888726
  },
  {
    "disk_usage": 6502878483,
    "time_boundaries": [
      "2024-03-15T21:35:41.384Z",
      "2025-04-08T01:32:28.825Z"
    ],
    "timeline": [
      [
        "2024-03-15T21:35:41.384Z",
        "2024-03-15T21:35:51.364Z"
      ],
      ...
      [
        "2025-04-08T01:32:18.845Z",
        "2025-04-08T01:32:28.825Z"
      ]
    ]
  },
  "/9j/4AAQSkZJRgABAQAAAQABAAD/2wBDAAgGBgcGBQgHBwcJC...
   ......................HvS/w0nf8KBH/9lkaGF2iQUCAA=="
]
```

Here, three methods are invoked: method `summary` for object `stor_106_0` on the interface `VideoStorage` (which ensures this is a video storage), method `channelSummary` on the same object with argument `"cam_106_3"`, and method `snapshotBase64` on the object with name `cam_106_7` (to get a base64-encoded JPEG data of a snapshot image from live stream from that camera. In HTTP response body respective results are returned as first, second and third elements of the array.

Another example:

```
$ curl http://192.168.32.106:8880/v1/svc -X POST \
    --data-binary '[["foobar", "barbaz"]]' | jq
[
  {
    "error": {
      "code": "NotFound",
      "reason": "Interface or method not found",
      "type": "NotFound"
    }
  }
]
```

This shows how the error can look like in the result from this HTTP endpoint: respective position in the resulting array is a JSON object in case of an error, with the property named

`error` non-null, set to either a string, or a form describing the error (in this case `NotFound`, because a non-existent method on a non-existent object was called).

The example of how this object can be called on the objects within a dynamically created cluster:

```
$ curl http://192.168.32.106:8880/v1/cluster/ClusterOne -X POST \
     --data-binary '[["cam1","snapshotBase64"],["webrtc0","getStatus"]]'|jq
[
  {
    "error": "Could not retrieve a snapshot: Timeout while getting a snapshot",
    "success": false
  },
  {
    "live": [],
    "sessions": 0
  }
]
```

– here, a live snapshot is requested from the camera `cam1` within cluster `ClusterOne`, and also the status of WebRTC server named `webrtc0` on that cluster is examined (see section 4.3.10 for more details on this flavor of API). The snapshot isn't returned because of an error (timeout), which is reported accordingly. The request for WebRTC server status is successfully returned, showing no currently active sessions.

# 4 Scripting and JS API

viinex 3.4 implements scripting by means of introducing an object of type `script`, every instance of which represens an independent JavaScript execution context. Section 2.5.3 explained how required source code in JavaScript is loaded into such context (that is – by means of specifying parameters `load` and/or `inline` in the configuration). In this chapter, the execution model for the script in viinex 3.4 is considered; the role of `onload`, `ontimeout`, `onupdate` and `onevent` handlers is explained[1]. In the section 4.2 the general JS API exposed by viinex 3.4 is documented. Section 4.3 goes into the detail of JS API provided by viinex 3.4 objects that can be controlled from scripts.

## 4.1 Execution model and handlers

Like it was mentioned in section 2.5.3, scripts serve for the following purposes in viinex 3.4:

- maintain internal state according to a custom logic, and expose a part of that state via HTTP API (see section 3.17.1);
- accept requests to update the internal state and reply to such requests to the parties who initiate them (see section 3.17.2);
- receive and process events from other objects; generate and send new events;
- query and control other objects like video recording controller, PTZ device, video renderer, stream switch, and so on, – by means of calling respective JavaScript methods for such objects (see section 4.3).

The update requests, as well as the events, targeted by two of those four purposes, are both initiated by an external party (a client who issued a HTTP request or an object which has generated an event). They can be triggered at any moment, independently of which stage of execution the script is in. On the other hand, JavaScript execution context is single threaded (at least in ECMAScript 5.1 [23], and there is no way in ECMAScript to preemptively interrupt a current execution of arbitrary code in order to execute some other portion of code in the same context, and the return the control to the original code.

This means that in order to handle requests or events initiated by external parties, and to do that timely, the script needs to be idle most of the time. This is because the idle state is the only state when the script is ready to process a new request or event.

The model of execution of scripts in viinex 3.4 does not resemble the typical program in C, which does not return control until its completion. Rather the script is an event-driven program, having a limited number of entry points which periodically trigger the execution of a portion of code in the script, – but the latter tries to do its job as fast as possible, and return the control – that is, get back to an idle state. Then some other external party initiates some

[1] For the purpose of this chapter the feature of overriding handlers' names is ignored: throughout this chapter the default names will be used.

action, triggering the execution of the script again. Good thing is that the execution context between the moments when the script execution is triggered over and over, is preserved for the same instance of script. This means that the script may "remember" its history of execution, if it wants to.

In viinex 3.4 there are four entry points which are used to trigger the execution of a script. These entry points are called handlers, and their default names are: `onload`, `ontimeout`, `onupdate` and `onevent`. Their signatures, syntax and semantics are covered in the next subsections. It is important that none of these handlers should retain execution control for a long time. Instead, each handler should do its job as fast as possible, and return. The script will gain control again when the next event or HTTP update request occurs, and if the script knows it needs to gain soon unconditionally – it can schedule a timer for itself.

### 4.1.1 `onload`

The `onload` handler is called exactly once, when the script execution context is ready for normal operation – that is, all source code of the script is loaded, and all viinex 3.4-specific objects and API is injected into the JS execution context.

`onload` should be a function of one argument. The argument that it receives is the value of `init` parameter of this script instance in viinex 3.4 configuration, and is typically used as the initial configuration for the script – that is, the configuration specific for the JS code.

### 4.1.2 `ontimeout`

The JS API provides the means for the script to schedule a timer, in order be sure that the control will be gained by the script again within certain amount of time, – regardless of whether events or update requests are received. The JS API function which schedules such timeout is called `vnx.timeout`, see paragraph 4.2.2 for details.

The `ontimeout` function is called when the timer previously scheduled by the script rings. This handler has no arguments.

### 4.1.3 `onevent`

The `onevent` handler is called when the instance of `script` object receivs an event from other object that it is linked with in viinex 3.4 configuration.

`onevent` should be a function with one argument, which receives the JS object (structure) containing the event itself. Event objects that are passed as arguments of the `onevent` handler have the following form:

```
{
  "timestamp": TIMESTAMP,
  "origin": {
    "name": STRING,
    "type": STRING,
    ...
  },
  "topic": STRING,
```

```
  "data": OBJECT
}
```

– just like an external JSON representation of an event sent via WebSocket interface, see section 3.18.

### 4.1.4 onupdate

The `onupdate` handler is called when a request for "update" is received via the HTTP API. For more information on the client side representation of that API see section 3.17.2.

An argument to the `onupdate` function is the data sent by an HTTP client as update request body (in JSON format). HTTP server parses the stringified JSON syntax, converts it into internal representation, and passes as an argument to the `onupdate` function.

Among all other handlers, `onupdate` is the only one for which its return value is important. This value is serialized into JSON format and passed back to the HTTP client as a response to his initial update request.

### 4.1.5 Example

To sum up the information on handlers in viinex 3.4 scripts, an annotated example of handlers test implementation is given below. The script shown as an example counts the number of events it has received. It also periodically sends an event of its own. For that, it schedules the timer for itself, and uses the timeout handler to update its state and send an event. The script also accepts the update requests via HTTP to set the new value of timeout interval for its periodic events.

```
// file test-script.js
var config;

var state;

function onload(conf){
    if(conf && conf.interval)
        config = conf;
    else
        config = { interval: 5 }; // default interval is 5 seconds

    state = {
        events: 0,
        motion: 0,
        timeouts: 0
    }

    // initially publish our state for HTTP GET calls
    vnx.publish(state);

    vnx.timeout(config.interval); // schedule an initial timeout
}
```

```
function ontimeout(){
    state.timeouts = state.timeouts + 1; // update state
    vnx.publish(state); // publish our renewed state

    // send a test event
    vnx.event("TestTopic", { timeouts: state.timeouts });

    // schedule the next timeout
    vnx.timeout(config.interval);
}

function onevent(e){
    state.events = state.events + 1;

    // also separately count motion alarm events
    if(e.topic == "MotionAlarm" && e.data.state)
        state.motion = state.motion + 1;

    vnx.publish(state); // publish our renewed state
}

function onupdate(d){
    if(d && d.interval){
        // update the interval
        config.interval = d.interval;
        // reschedule the next timer
        vnx.timeout(config.interval);
    }

    // return some information to the HTTP client
    return {
        hello: "world",
        state: state,
        config: config
    };
}
```

Consider this script is deployed with the following configuration (note the `script` section and the `init` property in it, and how it is used in the `onload` handler in the script):

```
{
    "objects":
    [
        {
            "type": "onvif",
            "name": "cam1",
            "host": "192.168.0.121",
            "auth": ["admin","12345"]
        },
        {
            "type": "webserver",
```

```
            "name": "web0",
            "port": 8880,
            "staticpath": "share/web"
        },
        {
            "type": "script",
            "name": "script1",
            "load": ["test-script.js"],
            "init": {
                "interval": 5
            }
        }
    ],
    "links":
    [
        ["cam1", "web0", "script1"]
    ]
}
```

This script can be tested using the CURL utility:

```
$ curl -X GET http://localhost:8880/v1/svc/script1
{"events":41,"timeouts":5,"motion":9}

$ curl -X POST http://localhost:8880/v1/svc/script1 --data '{"interval":1}'
{"state":{"events":86,"timeouts":8,"motion":23},"hello":"world",
"config":{"interval":1}}

$ curl -X GET http://localhost:8880/v1/svc/script1
{"events":101,"timeouts":17,"motion":23}
```

As can be seen, the counters increase over time. Update requests are accepted (and some custom response is returned for every such request), and the script's state change can be initiated by that requests too.

Also note that there is no handler for querying (reading) the script state. The script calls a special function `vnx.publish()` in order to publish its state, which can be later obtained by clients using HTTP GET request to that script. Once the state is published in this way, – viinex 3.4 automatically serves HTTP requests to get this state, and that does not require attention from JavaScript code.

### 4.1.6 Asynchronous operations and anonymous callbacks

Additionally, for some APIs, including the HTTP client described in 4.2.10, the operations are implemented asynchronously, which means that a script may be idle while the operation is being processed by viinex 3.4, – and then, upon completion, the result of the operation is returned to the script in form of an (asynchronous) call to the callback provided by the script. This does not mean that the execution of a script is interrupted, – quite the opposite, the script must be in idle state in order to be able to handle such callback. However, the callback is called when the operation is completed, and the script cannot control the asynchronous operation after one has started.

In viinex 3.4, the functions which are implemented as asynchronous operations, always accept callbacks with the following signature:

```
function (result, error) {
  ...
}
```

It is always the case that only one of the two arguments is passed when the callback is being called. If the operation has failed, the `error` argument is present, no matter what type of `result` is expected (there can be no result, – the error description is anyway passed to the callback in a second argument). Therefore it is advised that the callback checks whether its second argument is defined, and handles this case as a failure of the operation. The `error` argument is an object containing the boolean property `error` set to `true` and a description of the error in the string property `message`.

If no error has occured during the operation processing, the second argument passed to the callback is `null` or `undefined`, while the first argument holds the result of the operation. It depends on the operation and can be `null` as well in some cases.

The following example illustrates how the timeout can be scheduled in viinex 3.4 scripts using anonymous callback:

```
vnx.timer.delay(5.0, function(r,e){
    if(e){
        vnx.error("Operation failed: ", e.message);
    } else {
        vnx.log("Timer triggered successfully");
});
```

## 4.2 General puropose functions

In order to communicate with viinex 3.4 and other objects running in it, the JavaScript execution context is populated with a few specific functions and variables. They all are injected into "namespace" (top-level stateless JavaScript object) `vnx`. Most of these functions were previously disclosed in section 4.1.5, namely – `vnx.publish()`, `vnx.timeout()` and `vnx.event()` were used in the source code of the example. This section explains that viinex 3.4-specific functions in detail.

### 4.2.1 `vnx.publish()`

The function `vnx.publish()` serves to publish the "public view" of the state of the script. This public view is available by HTTP clients issuing the GET requests to the script, as described in section 3.17.1.

An argument to this function should be a JSON value representing a public view of the state of the script. Upon clients' request this JSON value is stringified and served as HTTP response body. This happens automatically, concurrently, without the actual participation of the script.

### 4.2.2 vnx.timeout()

`vnx.timeout()` is the function to schedule a timer for a script in order for the handler `ontimeout` to be called in specified amount of time. This is effectively the way for the script to gain control in specified time interval unconditionally, regardless of the presence of other triggers – events and/or update requests.

This function accepts one argument which should be an number of seconds (possibly fractional). The semantics of the timer in scripts is very simple: after a script calls `vnx.timeout(K)` and yields the control, the handler `ontimeout` of this script will be called `K` seconds later, exactly once. If a script needs the timer to be called again, – it should issue a new call to `vnx.timeout()`.

Also note that there is only one timer for each script, and each subsequent call to `vnx.timeout()` cancels the previous one.

There is also a way to cancel the timer (which might or might not be set previously), by calling `vnx.timeout()` with no arguments.

Note that there is also an alternative way to schedule a delay in viinex 3.4 builtin scripts, the one which uses anonymous callbacks. It is described in the next section 4.2.3.

### 4.2.3 vnx.timer.delay()

While the `vnx.timeout()` function and the `ontimeout` handler operate the single global timer available in the script, – there is another object named `vnx.timer` which is a factory for creating multiple timers and delays. There is one method available in that object, `delay(t,cb)`, which allows one to create a new delay. The first argument to this function is the value of timeout, in seconds, and can be a fractional number. The second argument to the `vnx.timer.delay()` is a callback function. This callback is called upon delay completion. The signature of the callback is described in section 4.1.6. There should be two argument in a callback function, where first one represents the asynchronous call result (in case of success), and the second one represents the error information (in case of failure). For the `delay()` asynchronous call, there is no async result, and the error is unlikely to happen, therefore a delay callback can actually ignore its arguments.

Note that the asynchronous delays created by `vnx.timer.delay()` do not affect each other and the global `timeout()/ontimeout()` pair. Correspondingly, their respective callbacks are triggered independently of each other and of the designated `ontimeout()` callback. A script may create as many concurrent asynchronous delays as needed.

The next fragment of code shows how periodical timers can be implemented using the mechanism of delays with anonymous callbacks:

```
function setDelay(x,d){
    vnx.timer.delay(d, function(r,e){
        vnx.log(x+"triggered, delay was ",d);
        setDelay(x,d);
    });
}

function onload(config){
    setDelay("A:",5);
```

```
    setDelay("B:",2);
}
```

The output of such script would be as follows (excerpt from the viinex 3.4 log):

```
...
[script.scr0/INFO] B:triggered, delay was 2
[script.scr0/INFO] B:triggered, delay was 2
[script.scr0/INFO] A:triggered, delay was 5
[script.scr0/INFO] B:triggered, delay was 2
[script.scr0/INFO] B:triggered, delay was 2
[script.scr0/INFO] A:triggered, delay was 5
...
```

As it can be seen, the concurrent delays are executed in parallel, not affecting each other.

### 4.2.4 `vnx.event()`

If a script is linked in viinex 3.4 configuration with other objects that implement the event consumer contract (these are, for example, objects of type `webserver`, `process`, etc.), it can generate and broadcast events to these objects. For that, the function `vnx.event()` is used.

This function expects two arguments. The first argument is mandatory, it should be of string type, and should contain the topic of an event to be sent. Topic can be an arbitrary string, but usually it is usually a short ASCII string with no whitespaces, resembling an identifier in C-like languages. There are predefined event topics in viinex 3.4 enumerated in section 2.4.3; generally they originate from ONVIF specifications. Applications are free to use this predefined event topics or introduce their own. The purpose of event topic is that it can be used by some event consumers to filter out irrelevant events. In particular, WebSocket interface provides the means for the client to subscribe for events of particular list of topics only. For more information on such subscription see section 3.18.

The second arument to the function `vnx.event()` is optional and may represent so called "event data". The format of event data depends on event topic. For instance, the events of topic `MotionAlarm` and `DigitalInput` have the event data of the form `{ "state": BOOLEAN }` – in this way they carry one boolean flag which describes the state of the motion detector or a digital input pin. An application is free to introduce its own forms for event data.

Note that the representation of an event in viinex 3.4 includes, besides event topic and event data, also information on when an event was produced (the field `timestamp`), and an information on event origin – the field `origin` which typically includes the object type where an event originates from, its name, and sometimes more: for instance, for digital input events there is also a number (an address, or an index) of a pin whose state has changed.

To avoid confusion, viinex 3.4 does not allow for scripts to fabricate these fields of an event[2]. When `vnx.event()` is called by the script, the event topic and data is set by the caller, however the timestamp of a newly generated event and its origin is set automatically by viinex 3.4. In case of script, if `vnx.event(TOPIC, DATA)` is called, the resulting event would have the form of

[2]The same policy is true for the `process` object described in section 2.5.4: an external process can specify the topic and the data of an event, but not the timestamp and origin.

```
{
    "timestamp": TIMESTAMP,
    "origin": {
        "type": "Script",
        "name": STRING
    },
    "topic": TOPIC,
    "data": DATA
}
```

where `timestamp` receives the value of current time according to the computer's clock, in UTC timezone, and `origin.name` receives the name (identifier) of this script instance in viinex 3.4 configuration.

### 4.2.5 Logging

The development and debugging of scripts in embedded environments is sometimes tough, and viinex 3.4 is no exclusion. There is no symbolic debugger for scripts, no breakpoints, variables cannot be watched at different stages of script execution, and so on. In order to ease the debugging and diagnostics of scripts, viinex 3.4 provides the means for logging. There is a family of functions, `vnx.log()`, `vnx.debug()`, `vnx.error()` and `vnx.warning()`, which allow the script to write a log records of different severity levels (`INFO`, `DEBUG`, `ERROR` and `WARNING` respectively). The records produced by said functions are directed into current viinex 3.4 log destination (a syslog, or a log file, or a standard error stream), and the `--log-level` policy applied to them.

All of these functions accept up to five arguments (this limitation contrasts with `console.log()` function and friends in modern browsers).

### 4.2.6 `require()` and modules

viinex 3.4 implements a basic support for JS modules for builtin scripting system. For that, the function `require()` is defined which can be used to load JS modules residing at predefined search paths, which is by default

```
Program Files\Viinex\share\js,
Program Files\Viinex\share\js\modules
```

on Windows, and

```
/usr/share/viinex/js,
/usr/share/viinex/js/modules
```

on Linux. On Windows, the actual search path depends on where viinex 3.4 is installed. The modules should be organized as ECMAScript 5 modules, i.e. each module should be represented by a single JS file, and that file should contain the code which eventually sets the `module.exports` variable to a single function or to an JS object, – whatever is expected by clients of that module. Some examples for modules for builtin scripts are shipped with viinex 3.4 distribution.

One important example and application for viinex 3.4 modules is the module named `vnx--script-instance`. It allows one to put the whole implementation of a custom script into another module, and load it, afterwards lifting the `onload`, `onupdate`, `onupdate` and `ontimeout` handlers which can possibly be defined in that implementation module, – to the root naming context (scope) of the script. This makes it possible to create the script instances in viinex 3.4 configuration which do not rely on the presence of JS source code in particular path defined in the configuration file, mentioning the source code files in the `load` property of the script, – but rather the code can be loaded by configuring the script like that:

```
{
    "type": "script",
    ...
    "inline": "require('vnx-script-instance')('script-impl.js');",
    ...
}
```

where `script-impl.js` is a substitute for an actual script implementation module name.

### 4.2.7 Linked objects

Probably the most important feature of scripts is that they can control other viinex 3.4 objects. Which objects, in particular, the script is allowed to control, – depends on which objects are linked with this instance of `script` object in configuration.

From the point of view of the script, however, it is still a challenge to know what objects should be controlled by that script, what are their names and types, because the script does not have direct access to viinex 3.4 configuration. There is a part of configuration which is available to the script, namely, an `init` property of the `script` object configuration section. One could use this section to inform the script on which objects it is linked with and should control. However the same information is also specified in the `links` section of the configuration, so placing it also into `script`'s `init` configuration property would be redundant and error-prone.

viinex 3.4 provides a script with the means to discover which objects that script can access, and what interfaces do that objects implement. Namely, there is a variable exposed under the name `vnx.objects`. That variable is a JavaScript associative array (dictionary). The string keys to that array are the names (identifiers) of viinex 3.4 objects, which are linked with this instance of `script` object, and which have types (or implement interfaces) controllable from JavaScript code[3].

The values contained in `vnx.objects` dictionary represent the objects linked with the script. Every of that representations is a JavaScript object, which contains:

- all interfaces implemented by corresponding viinex 3.4 object;
- all methods implemented by all interfaces of corresponding viinex 3.4 object.

This needs to be explained in more detail. Like with HTTP API, in JS API the case is that one viinex 3.4 object may implement more than one functional interface. In order to distinguish between functional interfaces, and to let the script know explicitly whether an object

[3]Some interfaces, like event source or event consumer, do not require specific methods on their implementation objects to be called from JavaScript code. Such objects, even if they are linked with the script in configuration, are not exposed to the script through the `vnx.objects` dictionary.

implements a given specific interface, – the interfaces in viinex 3.4 JS API are represented as JS objects, each containing all methods that this interface provides, having the predefined name specific to that interface, and – injected under that predefined name as a property of an object which implements that interface.

For example, assume there is an video renderer with the name `"rend1"`, linked with the script. Such script would then have an access to the variable `vnx.objects`, which is a dictionary, and that dictionary would contain the value with key `"rend1"`. Furthermore, the value `vnx.objects["rend1"]`, which is equivalent to `vnx.objects.rend1`, would be itself an object, which can be checked for whether it implements an interface `LayoutControl`, using the following test:

```
if(vnx.objects["rend1"].LayoutControl){
    var layctl = vnx.objects.rend1.LayoutControl;
    // perform actions with layctl
}
```

The JS object `vnx.objects.rend1.LayoutControl` is an implementation of interface `LayoutControl` provided by viinex 3.4 object `rend1`. This JS object contains methods for controlling the layout of the video renderer, – methods specific for the layout control, for instance the method `layout()`. The particular methods specific to each interface are described in section 4.3. At this point it is important that said methods can be called using the notation like

```
vnx.objects["rend1"].LayoutControl.layout(....);
```

For simplicity, all methods provided by all JS interfaces of an object, are also injected into this object, directly. This means that in the above example the method `layout` can also be accessed like this:

```
vnx.objects["rend1"].layout(....);
```

– note the omitted reference to the `LayoutControl`. This might create a point of confusion if there is a name clash between methods implemented in different interfaces, in case if some object is unlucky enough to implement such interfaces at the same time, – but this situation seems unlikely or rare, and it is still resolvable by referencing the particular interface of an object when a method with shadowed name from that interface needs to be used.

The syntax for using the resulting objects contained in the `vnx.objects` dictionary thus should look natural to the developers who is familiar with languages like C++, C# or Java. The resulting object has all methods of all implemented interfaces available directly in it (just like with multiple inheritance in C++ or multiple interfaces implemented by a class in C# or Java); there exists a way to test whether an object implements a specific interface ("`obj.InterfaceName != null`" as a substitute for syntax "`obj is InterfaceName`" in C# or "`dynamic_cast<InterfaceName*>(obj) != nullptr`" in C++, or "`obj instanceof InterfaceName`" in Java), and there is also a way to cast an object to a specific interface, possibly getting null if the interface is not implemented (that is – simply "`obj.InterfaceName`" as a substitute for syntax "`obj as InterfaceName`" in C# or "`dynamic_cast<InterfaceName*>(obj)`" in C++).

### 4.2.8 Configuration clusters

If the script has its configuration property `clusters` set to `true`, it receives an access to functionality for managing viinex 3.4 configuration clusters, identical to HTTP API described

in section 3.19.

In particular, an object named `clusters` gets exposed in the `vnx` namespace of script's JS execution context, and that object is populated with three methods: `enumerate`, `shutdown` and `start`.

- `vnx.clusters.enumerate()` takes no arguments and returns an array of string names of clusters currently being run in the viinex 3.4 instance.
- `vnx.clusters.shutdown(name)` serves for the purpose of stopping a configuration cluster. It takes one parameter – a string representing the name of a running configuration cluster. The function returns `null` upon success.
- `vnx.clusters.start(name, config)` initiates the creation of a configuration cluster. It takes two arguments, – the first one should be a string value for the name of the new cluster, and the second one should be a JS object representing the configuration of that new cluster. This function returns `null` upon success.

Note that the `config` parameter to the `vnx.clusters.start()` call should be a valid configuration for the viinex 3.4 cluster, – an object including the `objects` and `links` sections, with references to the auxiliary object `publish`, and so on. And example for a cluster configuration is given in section 2.5.6.

### 4.2.9 Local filesystem

A script has a limited set of functionality to access the local filesystem (local to the host or a virtual environment where viinex 3.4 instance is running). The corresponding API is exposed in the `vnx.fs` object and consists of the following functions: `unlinkSync`, `existsSync`, `renameSync`, `readFileSync`, `writeFileSync`.

The names of these functions mimick the names of their related counterparts in the API implemented by the popular Node.js platform, however the number and semantics of these functions' arguments may differ from Node.js implementation. Namely,

- `unlinkSync(name)` serves to remove a local file. It accepts one string argument – a path to the file to be removed. The function returns `null` upon success.
- `existsSync(name)` checks whether a file on a local filesystem exists. The function accepts one string argument – a path to the file which existence needs to be checked. The function returns a boolean value. Note that unlike Node.js this function does not work on directories (i.e. it checks for existence of a *file*).
- `renameSync(oldName, newName)` moves or renames a file on a local filesystem. This function accepts two string arguments, the path to an existing file and the new name or path. The function returns `null` upon success. This function does not work on directories.
- `readFileSync(name)` reads out the entire content of a local file which resides in path `name`, and returns it as a string value. Note that unlike the Node.js implementation, this function does not accept the encoding parameter; the encoding of the file being read is assumed to be UTF-8 in viinex 3.4 built-in scripts.

- `writeFileSync(name, data)` writes the string represented by second function parameter `data` into a file which resides in path `name`. If parameter `name` contains a path prefix and specified directories do not exist, – they will be created by the `writeFileSync` call. Note that unlike the Node.js implementation, this function does not accept the `mode` parameter; the destination file is always overwritten (data is never appended to an existing file, and an existing file is never preserved), and the encoding to write out the `data` always defaults to UTF-8.

If you find some important part of the API for filesystem management is lacking to build the logic required by your application, please contact Viinex team so that we could add the missing functionality for you.

### 4.2.10 HTTP client

A basic HTTP client is available to viinex 3.4 builtin scripts. The main purpose for providing HTTP client functionality within viinex 3.4 builtin scripts is to give users more flexibility in how viinex 3.4 communicates with other systems. For instance, a script may issue an HTTP request when some event happens in order to notify a third party system on that event. HTTP calls from scripts can also be used to control other systems, even including remote viinex 3.4 instances. The functionality provided by this client is by no means comprehensive and cannot compete with HTTP client implementations like, for instance, the one available for Node.js. However it should be sufficient for simple communication with many HTTP APIs and servers provided by a wide range of third party software.

To access the functionality of HTTP client built into viinex 3.4 scripts, an object `vnx.http` object is provided. This object has three functions exposed:

- `get(req, cb)` – the method to issue an HTTP GET request,
- `post(req, cb)` – the method to issue an HTTP POST request, and
- `request(req, cb)` – the versatile function to issue a request with any HTTP method.

As the matter of fact the first two methods are just a paricular case of the third one; for this reason the method `request` will be considered here in detail, and the differences for `get` and `post` functions are mentioned later in this section.

The method `vnx.http.request()` accepts two arguments, of which the first one describes the HTTP request which should be made, while the second one represents an (anonymous) callback to receive the result of HTTP request.

The first argument, – request description, – should be a JS object containing the following fields:

```
req = {
    url: STRING,
    method: "get" | "post" | "put" | "delete",
    headers: [ [STRING, STRING] ],
    body: null | OBJECT | STRING,
    expect: "raw" | "json" | "response"
}
```

Of all these fields only the `url` is mandatory and does not have a default value. The `url` should, obviously, specify a destination of the HTTP request.

The field `method` may specify one of 4 most typical HTTP methods, encoded as a lowercase string, – the name of that method. If not specified, the `"get"` value is assumed for the `method` property.

The field `headers` may specify additional HTTP headers to be passed along with the request. Those headers may specify authentication, set cookies, and so on. The headers are encoded as array of pairs of strings, where each pair is, in its turn, encoded as an array of 2 elements. If omitted, no additional headers are transmitted with the HTTP request.

The optional `body` property may hold the body of HTTP request that should be passed to the server. This property may either be `null`, or it can be a JS object, – and in that case that object is passed to the HTTP server in JSON format, – or it can be a string. If omitted, the `body` value of `null` is assumed, which means that no content is transmitted to the server in the request body.

In case if `body` property is a string, it is assumed that it is a "Latin1" pseudostring, see section 4.2.11 for more details. In that case the value of the string is transmitted as an array of bytes, where the value of each byte equals to the number of Unicode code point of respective character of the string. This allows to send arbitrary data without the need to introduce additional interfaces like `ArrayBuffer` (with losing some performance). Note that this requires the developer of a script to properly set the `Content-Encoding` header of the request.

The `expect` property specifies how the HTTP resonse should be decoded by viinex 3.4 before the provided callback is called. The possible values for `expect` field are `"raw"`, `"json"` and `"response"`. The value `"raw"` means that HTTP response body is returned as a string. The value `"json"` means that HTTP response body is decoded as JSON and returned as a JS object. The value `"response"` means that the low-level information on HTTP reponse is returned to the callback, – together with reponse code, headers, and the body in a raw form. This is described below in more details. If omitted, the `"raw"` value is assumed for the `expect` property, so the response body is returned as a string.

Note that there is a difference in HTTP client behavior depending on which response format is chosen for the callback. In case of `"raw"` and `"json"` formats, the callback does not have an information on what kind of error a server has returned, – so only the HTTP responses with code 200 are handled as successful in that case. All other HTTP response codes (except redirects, which are handled by the HTTP client automatically) are treated as an error.

This effectively means that if some kind of authentication scheme needs to be implemented, – the script should either provide all credentials with every request as HTTP headers, or it should specify the `expect` property as `"response"`, and analyze the HTTP response codes to send the credentials in case of respective errors (403).

In contrast, if the value of `expect` is set to `response`, – the HTTP client only treats as an error when the failure occured at the transport level (i.e. the HTTP request could not be completed/the result was not received). If this is not the case, – then the HTTP response with any response code is passed to the script as successful result. The script needs to analyze the response by its own.

The first argument to the `request()` as well as `get()` and `post()` calls may hold just a string value instead of an object with request description. In this case this string is interpreted as an URL where the HTTP request should be made to. Rest of request parameters are assumed to be default.

Now, the second argument to the `vnx.http.request()` method should be a callback function. As any other callback function in viinex 3.4 builtin scripts, it should accept two arguments, of which first represents an HTTP response result, in case of success, while the second on repsresents an error. More information on anonymous callbacks is available in section 4.1.6.

As already mentioned, depending on the value of the `expect` property of request description given to the `vnx.http.request()` call, the response argument may hold different types of values. That is either a sting or a decoded JSON object, or, if the `req.expect` property was set to `"response"`, this would be a JS object in the form of

```
response = {
    code: NUMBER,
    message: STRING,
    headers: [ [STRING, STRING] ],
    body: STRING
}
```

This object represents the HTTP response. The fields `code` and `message` represent an HTTP response code returned by the server, and the accompanying string message. The `headers` property hold the HTTP response headers, encoded as an array of pairs of strings (where each pair is encoded as an array of exactly 2 elements). The property `body` holds the HTTP response body sent by the server.

Note that in all cases, the HTTP client reads out the response in full, before the callback is called.

The functions `vnx.http.get()` and `vnx.http.post()` only differ from the function `vnx.http.request()` in that they use the respective HTTP method (GET or POST) when making a request.

Below some basic examples are given for the usage of the HTTP client in viinex 3.4 builtin scripts.

```
function onload(config){
    vnx.http.get("https://google.com/", function(r,e){
        if(e){
            vnx.error("An error occured: ", e.message);
        }
        else{
            vnx.log("Got a response, length=", r.length);
        }
    });
}
```

The above script could give the following output in viinex 3.4 log:

```
[script.scr0/INFO] Got a response, length=15503
```

Another example involving more complex request description:

```
function onload(config){
    var rq = {
```

```
        url: "http://demo.viinex.com/v1/svc",
        expect: "json"
    };
    vnx.http.get(rq, function(r,e){
        if(e){
            vnx.error("An error occured: ", e.message);
        }
        else{
            vnx.log("Got a response: ", JSON.stringify(r));
        }
    });
}
```

The above script could give the following output in viinex 3.4 log (somewhat shortened):

```
[script.scr0/INFO] Got a response: [["WebRTC","webrtc0"],["VideoStorage",...
```

The same, expecting a detailed HTTP response instead of decoded JSON value as the result to the callback:

```
function onload(config){
    var rq = {
        url: "http://demo.viinex.com/v1/svc",
        expect: "response"
    };
    vnx.http.get(rq, function(r,e){
        if(e){
            vnx.error("An error occured: ", e.message);
        }
        else{
            vnx.log("Got a response: ", JSON.stringify(r));
        }
    });
}
```

where the script output looks as follows (shortened and aligned):

```
[script.scr0/INFO] Got a response:
  {"body":"[[\"WebRTC\",\"webrtc0\"],...,[\"SnapshotSource\",\"camPenguins\"]]",
   "headers":[["transfer-encoding","chunked"],
     ["date","Mon,11 Jan 2021 11:02:21 GMT"],
     ["server","Viinex"],
     ["content-type","application/json; charset=utf-8"]],
  "code":200,
  "message":"OK"}
```

### 4.2.11 Auxiliary functions

The object `vnx.Encoding` is used to expose four functions allowing the user to encode and decode to and from UTF-8 and Base64 encodings:

```
stringBase64 = vnx.Encoding.encodeBase64(stringInputLatin1);
stringLatin1 = vnx.Encoding.decodeBase64(stringInputBase64);
stringLatin1 = vnx.Encoding.encodeUtf8(stringInputUtf8);
stringUtf8   = vnx.Encoding.decodeUtf8(stringInputLatin1);
```

In the above pseudocode, suffix `Latin1` of a string means that the input or result of a function is a pseudostring, consisting of characters from ASCII or Latin1 subset. The Unicode code points of such characters reside in range `0x00--0xFF`, thus, it can be convenient to have such `Latin1` strings as a naive substitute for an `ArrayBuffer`, which viinex 3.4 does not currently provide within its scripting environment. Such "Latin1" pseudostrings can be used, for instance, with the API to issue HTTP requests, particularly to send arbitrary raw data to the server.

In contrast, suffix `Utf8` denotes a conventional string with arbitrary characters. Finally, suffix `Base64` means that the input or result is a string consisting of characters from Base64 character set.

## 4.3 Application interfaces

The API available to the JavaScript code is not as comprehensive as the one provided via HTTP. One reason for this is that certain functionality which is available via HTTP API, like obtaining a video stream, – is simply irrelevant to the use cases of JS API: there is nothing can be done with a video stream in viinex 3.4 embedded scripts. Other reason is that the feature of scripting is relatively recent in viinex 3.4 (it has first appeared in December 2018), and is still evolving, catching up with its HTTP counterpart. Some application interfaces may be totally unsupported in JS API, from others there may be some methods missing. If you think certain functionality needs to be supported in JS API in the first place, – please contact viinex 3.4 team.

### 4.3.1 `LayoutControl`

JavaScript interface `LayoutControl` is intended for controlling the layout of a video renderer from scripts in viinex 3.4. This interface implements two methods,

- `sources()` and
- `layout(desc)`,

which mimick the behaviour of their counterparts in HTTP API of the layout contol interface, which is described in section 3.11. Method `sources()` does not accept arguments and returns a sorted list of string identifiers of live video sources linked to corrsponding instance of video renderer, just like the corresponding HTTP method described in section 3.11.1 does. The array of strings returned by this call should be used by the script to determine which index has a video source in this array, to use that index in layout definition.

The method `layout()` is used to set the layout of viewports displayed by the video renderer. It is a counterpart of HTTP API method for setting the layout for viewports displayed in a video renderer which is described in section 3.11.2. This method accepts one argument, which should be a JS object having the form described in section 2.8.5.

The example of using this interface in production can be found at https://github.com/gzh/viinex20-rlwswitchcfg/blob/master/v20_etc_conf.d/rlwswitch.js, see function `updateRenderers()` in that script.

### 4.3.2 PtzControl

The `PtzControl` application interface serves for controlling of the PTZ-enabled ONVIF compatible device from scripts in viinex 3.4.

`PtzControl` has the following methods implemented:

- `getDesc()`,
- `getPresets()`,
- `getPosition()`,
- `createPreset(name)`,
- `updatePreset(preset, name)`,
- `removePreset(preset)`,
- `gotoPreset(preset)`,
- `setHome()`,
- `gotoHome()`,
- `moveAbsolute(pan, tilt, zoom)`,
- `moveRelative(pan, tilt, zoom)`,
- `moveContinuous(pan, tilt, zoom)`, and
- `stop()`.

This interface exactly reproduces the functionality exposed over HTTP API, which is described in section 3.13.

### 4.3.3 RecControl

Interface `RecControl` allows the script to control the recording of a video stream into a video archive which is performed by recording controller, see section 2.2.2. Corresponding part of HTTP API is described in section 3.6.

The `RecControl` interface has three methods,

- `status()`,
- `record(b)` and
- `flush()`.

Method `status()` accepts no arguments and returns a boolean value signalling whether video recording is being performed. Method `record()` accepts one argument of boolean type which indicates whether the video recording should be started or stopped. Method `flush()`, just like its counterpart in HTTP API, forces the video archive linked with the instance of recording controller to complete the "current" video fragment, actually write all the data to the disk, along with the valid MP4 container headers, and immediately make that data available for reading via video archive API.

For example, the following code can be used to implement the video recording using recording controller `recctl0` whenever a motion detector is triggered, except business hours (say, 8AM–6PM):

```
function onevent(e){
    // process motion detector events only
    if(e.topic != "MotionAlert")
        return;

    // find out whether it's business hours
    var hour = e.timestamp.getHours();
    var isBusinessHours = (hour >= 8) && (hour < 18);

    // and whether it's an active or deactivated alarm
    var isAlarm = e.data.state;

    if(isBusinessHours)
        // on working hours, turn recording off
        vnx.objects.recctl0.record(false);
    else
        // otherwise turn recording on or off
        // if it's an active or deactivated alarm respectively
        vnx.objects.recctl0.record(isAlarm);
}
```

### 4.3.4 SnapshotSource

JavaScript interface `SnapshotSource` is indended for storing the snapshots from an object that provides the `SnapshotSource` interface, on a local filesystem. It is the mapping of endpoint `2.9.33` `SnapshotSource` and exposes two methods,

- `snapshotLatin1(source, options)`,
- `snapshotBase64(source, options)`, and
- `saveFile(destination, source, options)`.

The first one returns the value of type `string`, with characters each having its Unicode coding point number equal to the 8-bit unsigned integer value of a respective byte of JPEG data for the requested snapshot image. The second one provides the snapshot JPEG image data in Base64 encoding. The third method can be used to store the snapshot as a file on the locally mounted filesystem.

These methods take up to two or three arguments respectively, where the one named `destination` is mandatory and should be a string path to a file on the local filesystem. This file will be

created and will get the content of requested image in JPEG format. Note that it is required that the folder where the file should be created exists prior to the call to `saveFile()` method.

The argument named as `source` may represent a temporal requirement for the snapshot. It can be either a string containing a timestamp of requested image, or an integer number indicating an index of the requested image in the image cache of snapshot source. For detailed discussion on temporal requirements in snapshot requests see section 3.8.

The last argument `options` is optional all three methods and accepts a JavaScript dictionary which may contain any or both of the following properties: `roi`, in order to specify the region on the target image that needs to be saved as the snapshot, and `scale`, in order to specify the coefficient for spatial scaling when the image is saved.

Both of these parameter follow the semantics of respective parameters related to spatial requirements in snapshot requests described in section 3.8. The `roi` property, if given, should be an array of 4 elements representing the tuple of (left, top, right, bottom) coordinates of the rectangle that needs to be cropped from an original image. The coordinates are expected to be floating-point numbers within the range $[0, 1]$ each, – that is, relative coordinates normalized by image width (for left and right coordinates) and height (for top and bottom coordinates). Alternatively, the `roi` property may be an array of 4-tuples. This option can be used to crop more than one fragment from the original image. In that case the return value of the function will be an array of strings, each representing JPEG data of respective fragment of original image. This is specifically useful optimization for `channelSnapshot*` counterpart functions, where in order to provide a snapshot, a part of group of pictures has to be decoded, so if there is a need to crop several fragments of a video frame corresponding to some specific timestamp, such decoding will only be performed once.

The `scale` field of the `options` argument, if given, may be either a single number, which is interpreted as the coefficient on which the size of the original image, in pixels, is multiplied in order to obtain the result, or it can be a tuple of two numbers encoded as JS array, which is then interpreted as the target image size, in pixels, – so the image is cropped (if `roi` is specified), and after that it is downscaled or upscaled to fit to the size specified in "`scale:[width,height]`" parameter. Note that, as specified in section 3.8, the width and height are treated here as desireable values, however the actual size of resulting snapshot is chosen so that the aspect ratio of original image is preserved, and so that this resulting image with preserved aspect ratio fits the rectangle of a given size `[width, height]`. If the aspect ratio of the original image mismatches the aspect ratio equal to `width:height`, – then only one of dimensions of the bounding box would constitutes an "active" limitation, while the other would be redundant. One may also choose to fit just the width or just the height of a resulting image; this is achieved by specifying the `scale:[width,0]` or `scale:[0,height]` respectively. Then the snapshot is scaled so that the resulting image has specified width (or height), and the original aspect ratio is preserved. The same logic is applied when requesting the snaphot using HTTP API.

### 4.3.5 `Stateful`

The `Stateful` interface exposed via JavaScript is the counterpart of respective HTTP interface described in section 3.17.2, and is the mapping of endpoint `2.9.34 Stateful`. The purpose of this interface is an abstraction for some object which can provide some information on its publicly readable state. The interface has one method

- `read()`

and accepts no arguments. The implementation may return the requested information as a

result from the `Stateful.read()` call.

### 4.3.6 `StreamSwitchControl`

The purpose of JavaScript interface `StreamSwitchControl` is to control the stream switch object described in section 2.4.2. This interface exposes two methods,

- `sources()`, to enumerate possible input sources; and
- `input(n)`, to switch output between inputs.

Corresponding part of HTTP API is described in section 3.12.

Method `sources()` accepts no arguments and has the sematnics equivalent to that of synonumous method for the `LayoutControl` interface: it returns a sorted list of string identifiers of live video sources linked to corrsponding instance of the stream switch object. The returned list should be used by the script to determine which index has a given video source in this list, to use that index in order to switch output video stream to a specified input video stream.

The `input()` method accepts one argument, – an integer number which is interpreted as the index of an element in array returned by method `sources()`. Method `input()` makes the instance of `streamswitch` object to switch its output stream so that after the call it completed, the video stream from specified input source is passed through the `streamswitch`. This works exacly the same way as the HTTP call described in section 3.12.2 does.

The example of using this interface in production can be found at https://github.com/gzh/viinex20-rlwswitchcfg/blob/master/v20_etc_conf.d/rlwswitch.js, see function `updateVcams()` in that script.

### 4.3.7 `TimelineProvider`

The `TimelineProvider` interface is a mapping of endpoint `2.9.36 TimelineProvider`. It serves for the purpose of getting the table of contents of a video archive, typically implemented in an external third party VMS, on a specific video channel. This is described in more detail in section 3.7.6.

Within API exposed by `script` and `wamp` objects, this interface has one method,

- `timeline(begin, end)`,

which takes two optional parameters, `begin` and `end`, representing a temporal interval of interest for which the timeline should be returned. The return value of this method is an array of pairs, each encoded as an array of exactly two elements, where each pair represents an interval on the timeline where media data is present for the video archive channel (track) being requested.

### 4.3.8 `Updateable`

The `Updateable` interface is exposed via JavaScript by the same viinex 3.4 objects which implement the `Updateable` HTTP interface described in section 3.17.2 and is the mapping of

endpoint 2.9.37 `Updateable`. The purpose of this interface is an abstraction for some object which can accept commands for state modification. The interface has one method

- `update(val)`,

and accepts one JS value as its argument. The implementation may return a result from the `update()` method, that result is accessible by the calling script.

### 4.3.9 `VideoStorage`

The `VideoStorage` interface is the mapping of endpoint 2.9.40 `VideoStorage`. Its implementation matches the semantics descibed in section 3.5, except for paragraph related to media playback and exporting.

This interface implements six methods:

- `summary()`, to get overall information on video storage object instance;
- `diskUsage(begin, end)`, to get information on overall disk usage for a specified time interval;
- `channelSummary(channel)`, to get information on video data stored in the storage for a specified video channel;
- `channelDiskUsage(channel, begin, end)`, to estimate disk used by media data in the storage for a specified channel, in a specified time interval;
- `channelSnapshotLatin1(channel, source, options)`, to get the still image (snapshot) from the videoarchive of the requested channel (as a "Latin1" pseudostring containing JPEG data, see 4.2.11);
- `channelSnapshotBase64(channel, source, options)`, to get the still image (snapshot) from the videoarchive of the requested channel as JPEG data in Base64 encoding; and
- `channelRemove(channel, begin, end)`, to remove the data from video archive, when this is allowed by settings of the `storage` object and permissions of caller.

Each of these methods, except the `channelSnapshotBase64`, has its counterpart in the HTTP API and is described in respective sections: 3.5.1 for `summary()`, 3.5.4 for `diskUsage`, 3.5.2 for `channelSummary`, 3.5.3 for `channelDiskUsage`, and 3.5.7 for `channelRemove`. In all cases the mapping of HTTP request parameters to arguments of this API call should be obvious. The parameter `channel`, where present, identifies the video channel stored in the archive; parameters `begin` and `end`, where required, should be timestamps passed as strings in ISO format. The values returned by these methods exactly match the forms returned by their HTTP couterparts.

The method `channelSnapshotBase64` allows to get the still image (snapshot) data over the API from a video archive. As above, the `channel` argument should identify the video source within archive. The meaning of parameters `source` and `options` is described in sections 4.3.4 and 3.8 (parameter `source` here stands for temporal snapshot requirements, where `options` may hold the spatial snapshot requirements).

### 4.3.10 `WebRTC`

The `WebRTC` interface is a mapping of endpoint `2.9.44 WebRTC` and serves for the purpose of controlling a WebRTC server, including session creation and video playback. This interface matches the semantics, and data structures accepted and returned by its methods match the syntax described in section 3.14.

Interface `WebRTC` contains six methdos:

- `createSession(peerId, cmd)`, to create a new WebRTC session and possibly begin playback;
- `updateSession(peerId, cmd)`, to control video playback in an existing WebRTC session;
- `dropSession(peerId)`, to destroy an existing session;
- `setAnswer(peerId, sdpText)`, to set client's SDP answer when session is created;
- `getSessionStatus(peerId)`, to examine an existing WebRTC session status; and
- `getStatus()` to examine the status of the WebRTC server.

In all of the above methods but the last, `peerId` is a string uniquely identifying the WebRTC session within the server. This identifier is selected by client, and it's client's responsibility to keep session identifiers unique[4]

The parameter `cmd` for the `createSession` and `updateSession` methods should contain the request – an object of format described in section 3.14.3. Upon success, `createSession` returns a string value containing a SDP document, which can be passed to client to actually establish connection with the server. This is a so-called "offer" in WebRTC terminology.

In response, the client should generate and produce another SDP document describing the client's "answer". That response should be passed as a second argument to the call to `setAnswer(peerId, sdpText)` method.

The last two methods return the information on a specific WebRTC session or on the WebRTC server as a whole, in format described in sections 3.14.6 and 3.14 respectively.

## 4.4 Script-driven VMS integration API

This section does not introduce any new API functions, but rather defines the contract between viinex 3.4 and the script in the particular case when the script is the "driver" part of a script-driven VMS integration which is described (from users' perspective) in section 2.6.10. In other words, this section contains the information required to write new drivers for script-driven VMS integrations.

As it was mentioned in section 2.6.10, the object `scdrvms` should be linked with another object implementing endpoint of type `2.9.37 Updateable`, which is typically an object of

[4]Note that if the WebRTC server is linked with the authentication & authorization provider in viinex 3.4 configuration, then session identifiers need to be unique only for the authenticated principals, i.e. if different principals use the same session identifier, – that's actually different session identifiers. However this only makes sense when this API is used via WAMP interface, because the `script` object does not have an authenticated principal and, generally, won't be able to use this API with WebRTC servers which require authentication.

type `script`. In the script objects, the `Updateable` interface is implemented by means of method `onupdate()`. For script-driven VMS integrations this is used in the following manner: viinex 3.4 issues certain requests to the script, calling the `onupdate()` method with an argument of pre-defined form, and expects the return result from this method containing requested information, also in certain defined form.

Script-driven integration of other video management systems in viinex 3.4 has its limitations, unlike the integration via plugins API. To be specific, script-driven integration assumes that media is only retreived over RTSP, and other (RPC) interaction with VMS is done over HTTP. The protocol between viinex 3.4 and the driver script is designed with that limitations in mind. Typical implementation of driver script would construct URLs to let viinex 3.4 know where it needs to connect to, while network interaction itself is performed by viinex 3.4 rather than by script. The results of network interaction, are, however, processed (parsed) again by the driver script. In some cases (like the integration with Dahua NVR, shipped with viinex 3.4) the driver script may need to perform some additional interaction with the VMS over HTTP, in the background, – which is possible by using the HTTP client API described in section 4.2.10 of this chapter.

Below, the description of protocol between viinex 3.4 and the driver script is given. In order to help with real driver script implementations, viinex 3.4 comes with a helper JS module called `vms-driver`, which parses the input requests to `onupdate()` method and dispatches these requests to specific methods related to driver implementations. One can examine the source code of that module to find out what happens there; basically the difference boils down to the option of using separate functions for methods described above, and positional arguments to that functions instead of named properties of a single argument, as in case with `onupdate()` request. However for a custom script-driven VMS implementation the usage of that `vms-driver` module is optional; the information given below is sufficient to write such implementation without referring to additional code provided by Viinex.

### 4.4.1 Live media stream request

When requesting for a live RTSP stream, viinex 3.4 passes to the `onupdate()` method a JSON object of the following form:

```
{
    "method": "rtsp_live",
    "channel": OBJECT
}
```

Here property `method` is set equal to the string `"rtsp_live"` and property `channel` is set to the VMS channel selector according to the syntax described in section 2.1.5 (see syntax description for property `channel`).

In response, viinex 3.4 expects either a plain RTSP URL which should not contain credentials, or a JSON object containing the properties `url` for the RTSP URL of the stream, `auth` for the credentials, encoded as a JSON array of two strings, login and password, – and optionally the property `transport` which may enumerate preferred transport protocols to use for RTP transmission. These three options' syntax match that described in section 2.1.1.

Here is an example of how the code handling the request for a live media steram may look like:

```
function onupdate(req){
```

```
...
    if(req.method == "rtsp_live"){
        if(req.channel && req.channel.global_number === 42){
            return {
                url: "rtsp://192.168.0.121:554/cam42",
                auth: ["admin","12345"],
                transport: ["tcp"]
            }
        }
    }
...
}
```

Most likely the real code would contrstuct the URL using the parameters given at the startup of the script, combining them with the request parameters.

### 4.4.2 Archive media stream request

In order to get the information on archive media from the driver script, viinex 3.4 calls the `onupdate()` method of the latter with the JSON object value of the following form:

```
{
    "method": "rtsp_archive",
    "channel": ChannelSelector,
    "interval": [TIMESTAMP, TIMESTAMP],
    "speed": FLOAT
}
```

Here, property `method` is set equal to string `"rtsp_archive"`, property `channel` contains the VMS channel selector, property `interval` contains the temporal interval for which playback should be provided, and property `speed` contains a number indicating the desired speed of playback. The temporal interval is encoded as a pair of two strings, each of which may be parsed to obtain the ECMAScript5 `Date` object. Conventionally, viinex 3.4 operates with timestamps in UTC timezone. Depending on the underlying VMS, the driver script may need to convert them to a local (or any other) timezone.

As in previous section, the result of this call should describe RTSP source, that is it should either be a string containing RTSP URL without credentials, or a JSON object containing the same three properties `url`, `auth` and `transport`. In addition, that JSON object may also contain two more properties, `rangeStart` and `scale`. These two are converted into RTSP request headers that viinex 3.4 RTSP client would pass to the server along with `PLAY` request. The `scale` is converted to the value in `Scale` RTSP header and may be required to set it equal to the value of `speed` argument of `onupdate` request. The `rangeStart` property of response is converted by viinex 3.4 to `Range` RTSP header. The value of `rangeStart` property may be either set to a value of type `Date`, in which case it is converted to the form

```
Range: clock=TIMESTAMP-
```

where `TIMESTAMP` is substituted by value received from the driver, or the `rangeStart` property may be set to string literal `"now"`, in which case viinex 3.4 puts the following header into RTSP `PLAY` request:

```
Range: npt=now-
```

An example of how the `rtsp_archive` method may be handled is given below:

```
function onupdate(req){
...
    if(req.method == "rtsp_archive"){
        ... // checks on other arguments
        return {
            url: rtsp_prefix + "Streaming/tracks/" +
                makeChannelNumber(req.channel) +
                "/?starttime=" + format_rtsp_time(req.interval[0])
            ,
            transport: ['tcp'],
            auth: config.auth,
            rangeStart: "now",
            scale: req.speed
        };
    }
...
}
```

### 4.4.3 Get credentials for HTTP client

The above paragraphs describe calls to drver script for obtaining RTSP URLs. The next two paragraphs describe the information needed to get snaphots and the timeline from VMS being integrated. As previously mentioned, driver script provides viinex 3.4 with URLs to connect to, while the network interaction is performed by viinex 3.4 itself. Hence, the problem is to provide viinex 3.4 not only with URLs but also with credentials to perform an HTTP call.

The credentials for HTTP may be returned by the driver script to viinex 3.4 by handling `onupdate()` call with the argument set to a JSON object containing the property named `method` and equal to the string `"http_credentials"`. Currently the only possible response is the pair of strings, – login and password, – encoded as JSON array of exactly two elements. An example for such implementation is given below:

```
function onupdate(req){
...
    if(req.method == "http_credentials"){
        return ["admin", "12345"];
    }
...
}
```

Real-life example would probably use the configuration parameters to return from the driver script.

### 4.4.4 Get the snapshot image

When a snapshot is requested by user from viinex 3.4, the latter performs a call to the script driver's `onupdate()` function with an argument which is a ECMAScript5 dictionary of the

following form:

```
{
    "method": "get_snapshot",
    "channel": ChannelSelector,
    "request": {
        "when": null | Timestamp,
        "scale": Number | [Number, Number]
    }
}
```

Here, property `method` is set to the string `"get_snapshot"`, property `channel` is set to the channel selector, as described above for live RTSP URL request, and the property `request` contains the actual description of the request for snapshot. The value of `request.when` if set, denotes the desired timestamp of the image, or if not set – the request for a snapshot from live stream.

The property `request.scale`, depending on how the request was issued by user, may be represented by a single number (and then it's a requested scale factor with preserved aspect ration), or by an array of two numbers, and then it's optimal resulting image size. To preserve aspect ratio, one of the dimensions in size vector may be set to zero. An example for implementation of `get_snapshot` method is given below:

```
function onupdate(req){
...
        var request = req.request;
        var channel = req.channel;

        if(request.when){ // archive snapshot, not supported
            return null;
        }
        else{
            var resolution = "";
            if(request.scale && request.scale.length == 2){
                var width = request.scale[0];
                var height = request.scale[1];
                if(height == 0){
                    height=width*9/16;
                }
                else if(width == 0){
                    width = height*16/9;
                }

                resolution =
                    "?videoResolutionWidth=" + width +
                    "&videoResolutionHeight=" + height;
            }
            return api_prefix + "/ISAPI/Streaming/channels/" +
                makeChannelNumber(channel) + "/picture" + resolution;
        }
...
}
```

Note that the driver may elect to return `null` value from any path of the `get_snapshot` method implementation. The `null` value is interpreted by viinex 3.4 as instruction to fall back to the default implementation of getting the snapshot, which is to create a video stream, create video decoder, wait for a first decoded picture, encode it into JPEG and return the picture to the user.

Given that some VMS APIs may return something more sophisticated than just a JPEG image, another processing step is introduced, namely the "method" `snapshot_parse`:

```
{
    "method": "snapshot_parse",
    "body": STRING
}
```

(just like with other methods, the structure of this form gets passed as an argument to the `onupdate` function of the VMS driver script).

The `snapshot_parse` method of the VMS driver script is only called if the body returned in response to HTTP request described by `get_snapshot` call – is not recognized by viinex 3.4 as neigher raw JPEG image, nor a base64-encoded image with the "data URL" prefix, that is the fixed string `` which prepends the base64-encoded JPEG image. As an example, the endpoint described by the `get_snapshot` call may return a JSON structure containing the image encoded in some form. Function `snapshot_parse` may transform this kind of response body into an actual image. Like with `get_snapshot`, the `snapshot_parse` may return either a JPEG image body, and in that case it should be a "Latin1" pseudostring (see section 4.2.11), – or it should be a data URL containing the JPEG image, which is a base64-encoded JPEG prepended with the fixed substring ``.

An example of how the `snapshot_parse` method may be implemented is provided in the driver `vms-viinex.js`, which is shipped with viinex 3.4. Under the hood, one of the branches within `get_snapshot` function returns the description for requesting the scripting-style API endpoint (see section 3.20) on a remote viinex 3.4 instance to get a live snapshot of a video channel. That would be a call to `snapshotBase64` method, and its result represents a base64-encoded image, although without the data URL prefix, packed in a simple JSON data structure (array). The `snapshot_parse` function in that case is used to extract the string with base64-encoded image from that JSON structure, prepend it with the data URL marker, and return so that viinex 3.4 can recognize such result as an image.

### 4.4.5 Obtaining the timeline

The protocol for obtaining time timeline from the driver script has two variants: the one when the timeline can be obtained from the server in one HTTP request, or the one when the timeline needs a series of HTTP requests performed iteratively. Hence, unlike the above functions, the feature of obtaining the timeline involves several distinct calls to the driver script.

#### Timeline request initiation

Which variant should be used is defined by the driver script in the first call, which is `onupdate()` with an argument of the form

```
{
```

```
    "method": "timeline_get",
    "channel": ChannelSelector,
    "interval": null | [TIMESTAMP, TIMESTAMP]
}
```

Here, `method` is set to the tag value of `"timeline_get"`, `channel` contains the value of channel selector, and `interval` may contain `null`, which indicates that the whole history on media recordings was requested by user, or it may contain a pair of timestamps – begin and end points on time scale, within which the timeline information is requested.

The response to this call may have the form of

```
result = UrlString | HttpRequestObject
       | [ COOKIE, UrlString | HttpRequestObject ].
```

The first two options are sematically equivalent, the `UrlString` form is a trivial case for `HttpRequestObject`, – a JSON object which describes the HTTP request that viinex 3.4 needs to perform in order to retreive timeline information from the target VMS. The syntax and semantics of `HttpRequestObject` values are described in section 4.2.10.

#### One HTTP request scenario

If one of the first two forms are returned from `timeline_get` request, i.e. when return value is not an array, – viinex 3.4 is instructed that timeline may be obtained in just one HTTP call. viinex 3.4 then performs that HTTP call described by the result of `timeline_get` method, and once the HTTP response is obtained from the VMS, – the body of response is passed to the driver script method `timeline_parse`, which is the `onupdate()` call with an argument in the form of

```
{
    "method": "timeline_parse",
    "body": STRING
}
```

Here property `body` contains the body of HTTP response obtained from the VMS. The driver script should parse this data and return the timeline for the original request. The timeline should be returned according to the syntax described in section 3.7.6, that is – it should be represented as a JSON array of pairs of timestamps, each pair encoded as JSON array of exactly two elements. Each pair in the resulting array means the temporal interval for which the media data is present within the VMS being requested. After the `timeline_parse` call is completed, the original timeline request is considered completed as well.

#### Iterative scenario

However if the `timeline_get` request returns the result in the third form, that is

```
timeline_get_result =
    [ COOKIE, UrlString | HttpRequestObject ],
```

then viinex 3.4 is instructed that the timeline request should be executed iteratively. The first value `COOKIE` returned from `timeline_get` request may be an arbitrary JSON value chosed by the driver script, and has the meaning of opaque internal identifier of the request. It allows the driver script to keep the state of timeline requests being executed in parallel. The second value in the resulting array has the same meaning as for the first case, – it describes the HTTP request that needs to be performed by viinex 3.4 to obtain the timeline data.

The difference from the first case is that viinex 3.4 prepares to iteratively execute more than one of such request. It takes the following course of action:

1. the HTTP request description obtained from `timeline_get` request is considered "current";
2. the "current" HTTP request is executed by viinex 3.4. The result is fetched from the VMS;
3. viinex 3.4 calls the driver script `onupdate()` function with a special method `timeline_get_next` (see below), passing the timeline request identifier (`COOKIE`) and the HTTP response body obtained from the VMS;
4. if `timeline_get_next` returns the non-null value, that value is considered a HTTP request description, that description is considered "current", and the algorithm proceeds from step 2;
5. otherwise, if `timeline_get_next` returns `null`, viinex 3.4 finalizes this timeline request by calling method `timeline_get_collect`, passing to it the timeline request identifier (`COOKIE`). The value returned from `timeline_get_collect` is considered the result of timeline request, and is returned to the user.

The abovementioned methods `timeline_get_next` and `timeline_get_collect` are the calls to `onupdate()` function of the script with the following forms:

```
{
    "method": "timeline_get_next",
    "cookie": VALUE,
    "body": STRING
}
```

(`cookie` contains timeline request identifier, and `body` contains the body of HTTP response previously performed by viinex 3.4), which should return either a new HTTP request description to continue iterations, or `null` to proceed to the finalization step. The other one looks like

```
{
    "method": "timeline_get_collect",
    "cookie": VALUE
}
```

and should return the timeline request result of the form described in section 3.7.6.

The examples for timeline retreival implementations may be found with viinex 3.4 distribution in JS modules `vms-viinex` for the first scenario (where timeline is obtained in one HTTP request), and modules `vms-hikvision` and `vms-dahua` for the second scenario, where the timeline is being built iteratively.

# 5 WAMP interface

The WAMP client object, whose configuration is described in section 2.5.7, is used to expose the functionality of other viinex 3.4 objects so that it can be used by external software, by calling methods over RPC or subscribing for viinex 3.4 events.

This is very similar to how objects' functionality is exposed by a `script` object. The obvious difference is that in case of `script`, the functionality of other objects can be used by the code of that script (although it can be invoked over HTTP or by other means), – while in case with the `wamp` client, the functionality of other viinex 3.4 objects can be used by external software. Despite the difference, the objects of types `script` and `wamp` share common code for publishing other viinex 3.4 objects' API. That means that WAMP client does not invent its own API mapping, – it literally follows the API descibed in section 4.3. There is a difference in procedure naming convention taken by `wamp` object, however this difference boils down to the change of case of identifiers: the names of interfaces and methods are automatically converted to a so-called "snake case". The details are descibed below.

The other difference is that, while the source code of the `script` is installed locally, and script can only do what is implemented in its source code, – the `wamp` object exposes the API of other objects over network, – and for that reason it should be aware of security policies applied to the viinex 3.4 instance. This is additionally complicated by the fact that WAMP is a mechanism for *routed* RPC, so the callee does not have a session information to identify the caller. viinex 3.4 suggests to solve this by passing the security context as an explicit argument with each procedure call.

Besides the API exposed by viinex 3.4 objects linked to the `wamp` client, – the latter also publishes the events emitted by other objects linked to it. Current limitation (as per September 2021) is that this only happens if the WAMP client is not linked with an object of type `authnz`, and thus no security policies are applied.

## 5.1 Naming convention

The API exposed over WAMP and the events published within the router all should have, in terms of WAMP specification, so called "resource identifier", the URI. The specification mandates that such URI are formed in syntax similar to Internet domain names, but in a reversed order (i.e. most common part of domain name comes first). The URIs within a single WAMP realm form a common namespace.

For the implementation of `wamp` object in viinex 3.4, the prefix of WAMP URI which is used to expose the API and publish events can be specified in the configuration of that object, as described in section 2.5.7. Respective settings are configured by properties `app` and `prefix`. If, for example, the value of `app` is not set (then the default value of ``com.viinex.api'' is used), and the value of `prefix` property is equal to `wamp0`, then the API exposed by this instance of `wamp` object is going to have the prefix of

```
com.viinex.api.wamp0,
```

and so would have the events published by this object.

The structure of API exposed by the `wamp` client depends on which application objects are linked to it, and matches the API descibed in section 4.3. Each linked object's interfaces are exposed under the URI of `app.prefix.object_name.interface_name`, and each method is published under respective interface name. For brevity, methods are also re-exported directly under object name (i.e. without interface name). Currently this does not create naming conflicts, however in case of conflicts they can be resolved by using fully qualified method name (that is `app.prefix.object.interface.method`).

The difference from API described in section 4.3 is that, in order to match so called "strict URI" convention taken in WAMP specification, viinex 3.4 converts the names of objects, interfaces and methods from a so-called "camel case" to a so-called "quiet snake case". This conversion is performed automatically, and the rule of the conversion is — whenever a change of register occurs in an identifier, i.e. a pair of characters, an uppercase, then a lowercase, or vice versa, a lowercase, then an uppercase, is found, then a capital letter in such pair is prepended with an underscore. (The underscore itself in the original string and the digit characters are counted as a lower-case for the purpose of this transform). Then all the characters are converted to lower case. Afterwards, all sequences of one or more underscores in the middle of identifier are reduced to one underscore (for each sequence), and all prefix and suffix underscores are trimmed. For example:

```
Updateable   -> updateable
VideoStorage -> video_storage
WebRTC       -> web_rtc
Camera1      -> camera1
CAMERA1      -> camer_a1
PTZControl   -> ptz_control
foo_Bar      -> foo_bar
APP_HOME     -> ap_p_home
_A__b__c_D__ -> a_b_c_d
cam16hall    -> cam16hall
Cam16Hall    -> cam16_hall
```

Thus, the resulting methods could have the URIs of

```
com.viinex.api.wamp0.stor0.video_storage.summary
com.viinex.api.wamp0.stor0.channel_summary
com.viinex.api.wamp0.cam2.snapshot_base64
```

and so on.

Note that this conversion is only applied to WAMP URIs. The identifiers of objects within JSON structures transmitted over WAMP are left intact.

## 5.2 Calling convention

WAMP suggests that an RPC call may have positional and named parameters. Since the API exposed by an object of type `wamp` in viinex 3.4 matches the API which could be exposed by an object of type `script`, and the latter is a script in JavaScript (ECMAScript5) language, – viinex 3.4 API only utilizes positional parameters. Each RPC call to a `wamp` object should have

a respective number of positional parameters all passed as a single JSON array. (Depending on client language and the library which is used for interfacing WAMP, this marshaling may or may not happen automatically).

Same logic is applied for return values: since the API is designed to be native for JavaScript, – all methods return at most one value, which is marshalled by WAMP as a "first positional return parameter" (i.e. an array is returned by all API methods, an there is always at most one argument in that array).

The naming parameters, however, should be used in case if the `wamp` object is linked with an object of type `authnz`, which means that authentication is required to perform RPC calls. In this case, the client needs to authenticate against viinex 3.4 server and get the authentication token in JWT format – that would be a string. Respective calls are described in section 5.3. The resulting token then needs to be passed to the viinex 3.4 server with every RPC call as a named parameter to WAMP call requests. The name of respective parameter is `auth`. Here's how a call requiring an authentication may look in a `wamp-cli` tool:

```
session.call('com.viinex.api.wamp0.stor0.channel_summary',["cam1"],
{"auth":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJ1c2VyMSIsInJ
vbGVzIjpbInZpZXdlciJdLCJuYmYiOjE2MzE4OTI5MzksImV4cCI6MTYzMTkyODkzOX0.
jurelituK7_SKWi2L6RVkamDI-IKnXdvgBA3baIwDic"}).then(console.log)
```

Here, `com.viinex.api.wamp0.stor0.channel_summary` is an endpoint (remote procedure) being called, ``cam1'' is a (positional) call argument, and the string of ``eyJh...Dic'' is the authentication token passed in the named parameter `auth`.

## 5.3 Remote calls

### Authentication

There are two methods exposed by object of type `wamp` providing means for getting an authentication token: a request for challenge, and an authentication response.

The method to request for authentication challenge has the name of `auth.challenge` and accepts one parameter of type string, – the name of user (principal) who desires to be authenticated. The return value from this method is a JSON structure matching that described in section 3.2.1.

Here's an example of how this call may look like, taken by means of the `wamp-cli` utility:

```
$> session.call('com.viinex.api.wamp0.auth.challenge',
     ["user1"]).then(console.log)
{
  signature: '45b5b5ee4e067f2cb2d3335b20ea9606',
  when: '2021-09-18T13:12:53Z',
  realm: 'viinex',
  challenge: 'c2b...c1b'
}
```

The method to get an authentication token from the server is called `auth.response`, and it takes one argument – a JSON object with fields `login, when, challenge, signature` and

`response` filled in. The first four fields should match same fields returned by the `challenge` call, while the field `response` should be computed according to algorithm described in section 3.2.2.

Upon success, the `response` method returns a string representing the authentication token in JWT format. This token needs to be further used in all subsequent WAMP calls as a named parameter `auth`.

An example of how the `response` call may be performed is given below:

```
$> session.call('com.viinex.api.wamp0.auth.response',[
{
  "login": "user1",
  "when": "2021-09-18T13:35:26Z",
  "challenge": "f34...54c",
  "signature": "1a2051c1761ff4201a8917e6bdf86726",
  "response": "fd0b50d63ae35f72e122f30a2b240327"
}]).then(console.log)

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJ1c2VyMSIsInJvbGV
zIjpbInZpZXdlciJdLCJuYmYiOjE2MzE5NzIxNDcsImV4cCI6MTYzMjAwODE0N30
.gr7Y39Z32AvaGf6LuOfu3A5_dUbEdFgkG3lZLREPORc
```

### 5.3.1 Service interfaces

#### Enumerate endpoints and get objects' metainformation

Similar to HTTP API calls described in sections 3.1.1 and 3.1.2, WAMP client in viinex 3.4 exposes methods for acquiring the same information. Respective methods have the names of `svc` and `svc.meta`. Both of them do not accept aguments, except for the named parameter `auth` which should be used if authentication is required from a principal to access viinex 3.4 via the `wamp` client.

The `svc` call returns a JSON array of pairs, each encoded as a JSON array of exactly two elements, where the first element represents the type of an endpoint, according to section 2.9, and the second is the name of an object which exposes that endpoint. The example of this call could be as follows (in this example, an authentication token is passed to the server):

```
$> session.call('com.viinex.api.wamp0.svc',
     [],{"auth":"eyJh...ORc"}).then(console.log)
[
  [ 'WebRTC', 'webrtc0' ],
  [ 'VideoStorage', 'stor0' ],
  [ 'SnapshotSource', 'cam1' ]
]
```

The call `svc.meta` returns an array of pairs, each encoded as a JSON array of exactly two elements, where the first element is the name of an object, and the second element is the value of the `meta` parameter in respective object's configuration section. For example:

```
$> session.call('com.viinex.api.wamp0.svc.meta',[]).then(console.log)
```

```
[
  [ 'cam2', null ],
  [ 'cam1', { desc: 'Camera description' } ],
  [ 'stor0', null ],
  [ 'recctl1', null ],
  [ 'webrtc0', { stunsrv: 3478, 'rem stun': [Array] } ],
  [ 'web0', null ],
  [ 'authnz0', null ],
  [ 'wamp0', null ]
]
```

It's worth mentioning that the `svc` call, if made to an object which requires authentication, only returns the endpoints access to which is permitted, according to the calling principal's credentials. On the other hand, as `svc.meta` call operates with *objects* rather than endpoints, – and permissions are expressed in terms of endpoints, – such filtering is not performed for metainformation.

### 5.3.2 Application objects

As mentioned above, the application objects which are described in viinex 3.4 configuration and linked with the object of type `wamp` are made available for use by the latter by means of remote calls. These calls can be made according to the naming and calling conventions descibed in first two sections of this chapter. The endpoints (in WAMP terminology) to which the calls can be made have the form of `app.prefix.object.interface.method` or, the shorter form, `app.prefix.object.method`. The syntax and semantics of interfaces and methods is descibed in section 4.3 of the previous chapter. Some examples for possible calls to viinex 3.4 application objects over WAMP are given below:

```
$> session.call('com.viinex.api.wamp0.stor0.summary',[]).then(console.log)
{
  contexts: {
    cam2: { time_boundaries: [Array], disk_usage: 17596612179 },
    cam1: { time_boundaries: [Array], disk_usage: 30007065684 }
  },
  disk_free_space: 167623954432,
  disk_usage: 47603677863
}
```

The summary information for a video archive ``stor0'' is requested. The summary is returned in form described in section 3.5.1.

```
$> session.call('com.viinex.api.wamp0.recctl1.rec_control.status',
     []).then(console.log)
off

$> session.call('com.viinex.api.wamp0.recctl1.rec_control.record',
     [true]).then(console.log)
on
```

Here, the status of video recording is requeted from the recording controller ``recctl1'', and then the recording is turned on. In both cases the long form of endpoint name (including interface name) is used.

```
$> session.call('com.viinex.api.wamp0.webrtc0.get_status',
     [],{"auth":"eyJ...gcU"}).then(console.log)
{
  session_ids: [ '169985d3-2b70-4e9f-a5d7-540cd4da6958' ],
  sessions: 1,
  total_sessions: 1,
  live: [ 'cam2', 'cam1' ]
}

$> session.call('com.viinex.api.wamp0.webrtc0.get_session_status',
     ["169985d3-2b70-4e9f-a5d7-540cd4da6958"],
     {"auth":"eyJ...gcU"}).then(console.log)
{
  cookie: null,
  status: 'playing',
  last_frame: '2021-09-18T16:30:29.650958488884Z'
}
```

The status of a WebRTC server with name `webrtc0` is requested, and then the status of the WebRTC session is requested with the next call. Information is returned in the form according to sections 3.14 and 3.14.6 respectively. In both cases authentication token is passed to viinex 3.4 server. The user may only see and control its own WebRTC sessions on the server, although these sessions may be created by different means (i.e. one WebRTC session may be created using HTTP API, and then controlled over WAMP).

```
$> session.call('com.viinex.api.wamp0.cam1.snapshot_base64',[],
{"auth":"ey...ORc"}).then(console.log)

/9j/4AAQSkZJRgABAQAAAQABAAD/2wBDAAYEBQYFBAYGBQYHBwYIChAKCgkJChQODww...
...(a base64-encoded image returned as string)...4zRsDP/9k=
```

Here, a live snapshot image is requested from the object ``cam1''. The snapshot is returned as a string containing a base64-encoded JPEG image.

### 5.3.3 Managing configuration clusters

When the `clusters` property is set to `true` in the configuration of a viinex 3.4 object of type `wamp`, – that object exposes the API for controlling configuration clusters. That API matches semantics described in 4.2.8. The API is exposed under the name `clusters`, and contains three methods: `enumerate`, `shutdown` and `start`. In contrast with HTTP API, this API does not give access to the internals of the cluster(s). The reason for this is that this does not make much sense with a routed RPC which WAMP provides. Moreover, this would break the idea of routed RPC, because then in order to access some cluster, the client would need to remember where exactly that cluster is hosted. Instead, it is advised that another object of type `wamp` is created within every cluster, and these new `wamp` objects can be used to access other viinex 3.4 objects within their respective clusters.

An example of how a cluster in an existing viinex 3.4 instance may be created, removed, and how clusters may be enumerated, is given below:

```
$> session.call('com.viinex.api.wamp0.clusters.start',
     ["clusterOne", {}]).then(console.log)
[]

$> session.call('com.viinex.api.wamp0.clusters.enumerate',
     []).then(console.log)
[ 'clusterOne', 'main' ]

$> session.call('com.viinex.api.wamp0.clusters.shutdown',
     ["clusterOne"]).then(console.log)
[]
```

## 5.4 Events

When the object of type `wamp` in viinex 3.4 configuration is linked with other objects which expose endpoint of type 2.9.16 `EventSource`, – the former receives the events from respective objects and sends[1] these events to the WAMP router.

In order to enable clients for prefix-based and wildcard-based matching on topics, viinex 3.4 forms the topic of published events to have the following form:

```
app.prefix.origin.topic
```

where `app` and `prefix` are the values set in the configuration of `wamp` object, `origin` is the name of viinex 3.4 object which emitted the event, and `topic` is the topic of event in viinex 3.4 terms, trasformed to snake case, according to the rules described in section 5.1.

For example, WAMP events on motion detector alarm registered by viinex 3.4 object ``cam1'', retranslated by object `wamp0`, would have the topic of

```
com.viinex.api.wamp0.cam1.motion_alarm
```

This makes it possible for clients to subscribe for all events from all objects (using prefix matching on `com.viinex.api.wamp0`), on a specific viinex 3.4 object (using prefix matching on topic with object name included), or for viinex 3.4 events of a specific type using wildcard matching, like for example

```
session.subscribe('com.viinex.api...rtp_stats', onevent, {match: "wildcard"})
```

(that's not an ellipsis but three actual dots) to subscribe for all viinex 3.4 events of type `RtpStats` from all objects, and actually from all viinex 3.4 instances connected to the same WAMP realm.

The payload, i.e. the body of event, is sent by viinex 3.4 as the first positional parameter for the event, – and that parameter has the format of

[1] Currently, as per September 2021, there is a limitation: this only happens if the `wamp` object does not require authentication.

```
{
  "timestamp": TIMESTAMP,
  "origin": {
    "name": STRING,
    "type": STRING,
    ...
  },
  "topic": STRING,
  "data": OBJECT
},
```

described in more detail in section 3.18.

# 6 Native API

viinex 3.4 exposes a limited API for the low-level integration with its video subsystem. That API is contained in `vnxvideo` library which is published by Viinex Inc. under MIT license and is available at https://github.com/viinex/vnxvideo. The `vnxvideo` library is a part of viinex 3.4. The API of this library is mostly contained in files named `include/vnxvideo/vnxvideo.h` and `include/vnxvideo/vnxvideoimpl.h`.

The purpose of the `vnxvideo` library is a) to provide viinex 3.4 with access to certain low-level functionality related with video processing, and b) to provide other developers with abilities to extend viinex 3.4 so suite the needs of their applications.

## 6.1 Brief C and C++ API overview

The C API of `vnxvideo` library resides in the `vnxvideo.h` file. A brief excerpt from that file is given below:

```
typedef struct { void* ptr; } vnxvideo_videosource_t;
typedef struct { void* ptr; } vnxvideo_raw_sample_t;

typedef int(*vnxvideo_on_frame_format_t)(void* usrptr, EColorspace csp,
                                         int width, int height);
typedef int(*vnxvideo_on_raw_sample_t)(void* usrptr,
                                       vnxvideo_raw_sample_t buffer,
                                       uint64_t timestamp);

int vnxvideo_init(vnxvideo_log_t log_handler, void* usrptr, ELogLevel max_level);

int vnxvideo_local_client_create(const char* name, vnxvideo_videosource_t* out);

int vnxvideo_video_source_subscribe(vnxvideo_videosource_t source,
        vnxvideo_on_frame_format_t handle_format, void* usrptr_format,
        vnxvideo_on_raw_sample_t handle_sample, void* usrptr_data);
int vnxvideo_video_source_start(vnxvideo_videosource_t);
int vnxvideo_video_source_stop(vnxvideo_videosource_t);
void vnxvideo_video_source_free(vnxvideo_videosource_t);

int vnxvideo_raw_sample_get_format(vnxvideo_raw_sample_t,
                                   EColorspace *csp, int *width, int *height);
int vnxvideo_raw_sample_get_data(vnxvideo_raw_sample_t,
                                 int* strides, uint8_t **planes);
```

This API is actually is wrapper over the C++ interface. This approach is required to use the `vnxvideo` library via the FFI mechanism in languages with managed memory and garbage

collection. The C++ interface of `vnxvideo` library resides in the `vnxvideoimpl.h` file. Some excepts from that header are given below:

```
namespace VnxVideo
{
    class IBuffer {
    public:
        virtual ~IBuffer() {}
        virtual void GetData(uint8_t* &data, int& size) = 0;
        virtual IBuffer* Dup() = 0;
        // make a shallow copy, ie share the same underlying raw buffer
    };

    class IRawSample: public IBuffer {
    public:
        virtual void GetFormat(EColorspace &csp, int &width, int &height) = 0;
        virtual void GetData(int* strides, uint8_t** planes) = 0;
        virtual IRawSample* Dup() = 0;
        // make a shallow copy, ie share the same underlying raw buffer
    };
    typedef std::shared_ptr<IRawSample> PRawSample;

    typedef typename std::function<void(EColorspace csp,
                        int width, int height)> TOnFormatCallback;
    typedef typename std::function<void(IRawSample*,
                        uint64_t)> TOnFrameCallback;
    typedef typename std::function<void(IBuffer*, uint64_t)> TOnBufferCallback;
    typedef typename std::function<void(const std::string& json,
                        uint64_t)> TOnJsonCallback;

    class IVideoSource {
    public:
        virtual ~IVideoSource() {}
        virtual void Subscribe(TOnFormatCallback onFormat,
                        TOnFrameCallback onFrame) = 0;
        virtual void Run() = 0;
        virtual void Stop() = 0;
    };
    typedef std::shared_ptr<IVideoSource> PVideoSource;
    class IRawProc {
    public:
        virtual ~IRawProc() {}
        virtual void SetFormat(EColorspace csp, int width, int height) = 0;
        virtual void Process(IRawSample* sample, uint64_t timestamp) = 0;
        virtual void Flush() = 0;
    };
    typedef std::shared_ptr<IRawProc> PRawProc;

    class IH264VideoSource {
    public:
        virtual ~IH264VideoSource() {}
        virtual void Subscribe(TOnBufferCallback onBuffer) = 0;
```

```
        virtual void Run() = 0;
        virtual void Stop() = 0;
        virtual void Subscribe(TOnJsonCallback onJson) {}
    };
    // created by factory functions exposed from plugins

    VNXVIDEO_DECLSPEC IVideoSource *CreateLocalVideoClient(const char* name);
    VNXVIDEO_DECLSPEC IRawProc *CreateLocalVideoProvider(const char* name);
}
```

The above excerpt introduces interface classes for NAL unit buffer, for a raw video sample, for video source producer (`IVideoSource` and `IH264VideoSource`) and consumer (`IRawProc`).

There is also a number of auxiliary functions for managing video samples, deep copying them, creating shallow references, cropping, resizing, and so on. For more information please refer to the header file `vnxvideo.h`. If you need further help with that API, please refer to `vnxvideo` source code or contact viinex 3.4 support team.

In some cases the C API would be sufficient, like the use of "local transport" mechanism in order to acquire raw video from viinex 3.4 in some external process. In other cases, like H264 video source plugins implementation, it would be necessary to use C++ API. Depending on the context, various application problems are discussed below with the use of either one or another API.

## 6.2 Acquiring raw video by means of local transport

The local transport mechanism is a combination if shared memory (memory mapped files) and local IPC streams (named pipes on Windows or UNIX domain sockets on Linux) providing two parties, – local video provider and local video client, – with the ability to interchange raw video frames without the need to copy them. The implementation of that mechanism is open and is contained in `vnxvideo` project. It can always be used as a reference. Below, the particular use of the local transport mechanism is discussed to solve the problem of acquiring a raw video stream in an external process. The latter can be an arbitrary process running on the same host where the instance of viinex 3.4 runs; for example this can be an external process object managed by viinex 3.4 as described in section 2.5.4, but not necessarily: it can also be a standalone process like the `vnxview` program which serves to display a video stream on a screen; its source code is available at https://github.com/viinex/vnxvideo/tree/master/vnxview. Another example of use of a local transport is Viinex Virtual Camera[1].

The most important function for local transport clients is `vnxvideo_local_client_create`. This function creates a so-called local input video transport channel to receive raw video stream from a specified raw video source object with identifier `name` in viinex 3.4 running on a local computer. The underlying methods for acquiring raw video from viinex 3.4 process are shared memory (memory mapped files) and named pipes or UNIX domain sockets. In the calling process this function creates the respective IPC objects and returns an opaque pointer

[1]Viinex Virtual Camera is a DirectShow component installed with viinex 3.4 on Windows platform. This component implements the COM interface of a video source and is thus visible to other software as a locally attached webcamera. When an instance of Viinex Virtual Camera is created and used by a desktop application like Skype, Zoom, Chrome, Firefox, etc., – the implementation connects to a renderer object with a fixed name `rend0` which should be configured on the local viinex 3.4 instance. The video stream rendered by `rend0` object is then sent to each client connected to the virtual camera.

to the object that implements the `vnxvideo_videosource_t` interface, – respective pointer represents the local input video transport channel.

Like already mentioned, the latter allows the client code to subscribe for obtaining raw video frames and for the events of frame format change. This is done by means of the call to function `vnxvideo_video_source_subscribe`, which takes two callback functions that need to implemented by the client (see below).

The video source can be started and stopped. It is required to be started in order for the client code to receive the video data. Starting and stopping of a video source is performed with functions `vnxvideo_video_source_start` and `vnxvideo_video_source_stop` respectively. The video source should also be freed (disposed) when it is no longer needed, in order to prevent resource leak in the client's process. Such disposal is performed by means of a call to the function `vnxvideo_video_source_free`.

The raw video frames are passed to the client code by means of calling the callback functions provided by client into `vnxvideo_video_source_subscribe` call. These two callback functions should be implemented by client. Their types are `vnxvideo_on_frame_format_t` and `vnxvideo_on_raw_sample_t`. The latter receives the pointer `vnxvideo_raw_sample_t` for every video frame. This pointer allows the client code to get access to the actual uncompressed video data – for that purpose the function `vnxvideo_raw_sample_get_data` should be called. The out parameters of that function are 3 element arrays of integers (for video planes' strides) and pointers (for video planes' data).

The callback function `vnxvideo_on_raw_sample_t` should be aware of the lifetime of the video frames passed as its arguments. In particular, the pointers passed to this function are only valid till the function has returned control to its caller. If the application needs to store the video frame for a longer time, – it should either create a shallow reference to that frame, or make a deep copy of it. Both approaches have their pros and cons. Creating a deep copy of the video frame can be expensive, because it requires memory allocation and copying of up to several megabytes of data. However, after such deep copy is created, the application owns it and is free to hold that copy for as long as it is needed. Any number of frames can be copied in that way and held simultaneously, – the only limitation here is the amount of RAM. On the other hand, a shallow copy represents just a reference to a common buffer in memory, which holds the video data. The shallow copy (reference) creation is cheap for the video frame. The disadvantage of shallow copies of video frames received by a client application from the local input video transport channel comes from the fact that the latter uses shared memory to receive video from the viinex 3.4 instance. Such shared memory is a scarce resource, and "shared" here means that the instance of viinex 3.4 uses it too, and regularly needs to allocate memory buffers for the new video frames from the same memory pool. If that pool is exhausted, viinex 3.4 won't be able to produce a new raw video frame at the side of the video source. Such exhaustion can be caused by the client applications that store shallow references to many video frames for indefinite amount of time. In other words, sharing the video frames between multiple processes requires correct cooperative behaviour from that processes, namely – the client implementations should strictly limit the number of stored shallow references to video frames shared with viinex 3.4 instance.

## 6.3 Implementing the H264 video source plugin

In order to implement the H264 video source plugin for viinex 3.4, one should create a shared library containing a factory function which returns the instances of `IH264VideoSource` class. The complete example of such plugin implementation is available in the `vnxvideo` library in

file `FileVideoSource.cpp`. Below the most important issues for plugin implementation are considered.

The factory function to be exposed from the plugin shared library should have the signature of

```
typedef int (*vnxvideo_h264_source_create_t)(const char* json_config,
                                             vnxvideo_h264_source_t* source);
```

The first argument to this function is a string which contains serialized JSON value of `init` parameter of configuration (see section 2.1.3 for more details). Second argument of this function is an out-parameter and should be used to return the pointer to the plugin instance.

This pointer to plugin instance is used in viinex 3.4 by means of calling `vnxvideo_h264_source_*` family of functions (`*subscribe`, `*events_subscribe`, `*start`, `*stop`, `*free`). Default implementation of said functions is available in the file `vnxvideo.cpp`; as it can be seen, this default implementation treats the argument of type `vnxvideo_h264_source_t` as the pointer to an instance of C++ interface class `VnxVideo::IH264VideoSource`. Therefore, unless the `vnxvideo` library is not substantially modified to change these default implementations of `vnxvideo_h264_source_*` functions, – the plugin implementation should follow this convention. Namely, the factory function should return an instance of interface class `IH264VideoSource`.

This interface declares four methods: to start and stop the plugin, and to subscribe for video data and to subscribe for events. The latter is important if the plugin implementation needs to produce events synchronously with the video stream. Both subscription methods receive a callback functors. The plugin implementation should use these callbacks to pass the video or event data to viinex 3.4 for further processing.

> **NB!** A special care needs to be taken about timestamps related to media data, when producing that data from a plugin. Unlike with RTSP client, for plugin video source viinex 3.4 does not analyze frame or event timestamps generated by plugin, but preserves them. viinex 3.4 assumes that a live video stream is a sequence of frames with monotonously increasing timestamps. Moreover, for live streams it is assumed that the timestamp does not drift very far from the system clock. While for network-originated video sources viinex 3.4 performs such checks and corrections automatically, – for plugins it's the plugin author responsibility to fulfill these assumptions.

The `vnxvideo` library contains an example implementation of the `IH264VideoSource` class, namely – a plugin to read the content of a media file and produce the video stream (from the video track) and the sequence of events (from the subtitles track). The source code of this implementation resides in the file `FileVideoSource.cpp`; it can be used as a template for implementing other plugins.

## 6.4 Implementing the VMS integration plugin

An integration of a video management system (VMS) that is not yet integrated in viinex 3.4 can be added using the plugins mechanism. The configuration for respective integration is

described in section 2.6.9. The most important items in the configuration is the name of dynamically loaded library which contains the plugin implementation, and the name of entry point function which is called by viinex 3.4 in order to instantiate the VMS representation. This function should have the following signature:

```
typedef int (*vnxvideo_vmsplugin_create_t)(const char* json_config,
                                           vnxvideo_vmsplugin_t* vmsplugin);
```

The above function should be a factory, which should accept and parse the first parameter `json_config`, containing a configuration for connecting to the VMS.

The factory method should return a pointer to the C++ object (by convention, this pointer should be returned in an out parameter `*vmsplugin`, while the return value should be used to signal an error), which is supposed to implement the following class:

```
class IVmsPlugin {
public:
    virtual ~IVmsPlugin() {}
    virtual IH264VideoSource* CreateLiveSource(
                const CVmsChannelSelector& channelSelector) = 0;
    virtual IH264VideoSource*
            CreateArchiveSource(const CVmsChannelSelector& channelSelector,
                                uint64_t begin, uint64_t end,
                                double speed = 1.0) = 0;
    virtual std::vector<std::pair<uint64_t, uint64_t>>
            GetArchiveTimeline(const CVmsChannelSelector& selector,
                               uint64_t begin = 0, uint64_t end = -1) = 0;
    virtual IBuffer* GetSnapshot(const CVmsChannelSelector& selector,
                                 uint64_t timestamp) = 0;
};
```

Four methods in total need be implemented here: the first two serve as a factory methods for creating the live video source (`CreateLiveSource`) or a video source for accessing the video archive (`CreateArchiveSource`). The third method `GetArchiveTimeline` serves to acquire the contents of a video archive from the VMS for respective video channel and, optionally, for selected time interval represented by its begin and end points. The fourth method `GetSnapshot` serves to acquire a JPEG image from the VMS, containing a single frame from either a live stream.

The implementation of `GetSnapshot` method can be omitted (may throw an exception); in that case viinex 3.4 would use video stream factories to access the H264 data, transcode a frame to JPEG and use the resulting image as a snapshot. It is recommended though that the `GetSnapshot` method is fully implemented, because the VMS may often respond with a snapshot image more effectively than this can be done by transcoding a part of H264 stream.

# 7 Deployment

## 7.1 Installation

### 7.1.1 Windows

A Windows Installer database package (MSI) is provided for installing viinex 3.4 on Windows operating systems. The MSI package has an option for attended and unattended installs. The attended installation is has only one explicit step with only two options to choose: that is the default configuration file, and default log level.

**NB!** viinex 3.4 for x86 is only built to run on 64-bit operating systems.

viinex 3.4 comes with a number of predefined configuration files, however it is not necessary to use them. It is possible, while installing viinex 3.4 to make a choice to keep existing configuration file, even if it is customized. This is a typical option for the case is configuration is generated by embedding software.

The configuration file on Windows installations is stored in folder

```
Program Files\Viinex\etc
```

The installer places a number of predefined configuration files in that folder; which actual configuration file will be used is defined by the value of registry key

```
HKEY_LOCAL_MACHINE\SOFTWARE\Viinex\Viinex30\Config.
```

The installer sets this registry key to the value chosen by user (if ever) before the installation. Embedding software may overwrite this value at its discretion. This value may also point to a configuration directory rather than a single file; in that case, the split configuration logic described in section 2.13 is applied. It is possible to choose split configuration when viinex 3.4 is installed: there is a corresponding item "Configuration directory (conf.d)" in installer's GUI, and the folder for split configuration resides in

```
Program Files\Viinex\etc\conf.d
```

viinex 3.4 is by default installed as Windows service named "Viinex". It can be started and stopped with standard commands (`net start`/`net stop` or `sc start`/`sc stop`), as well as from Windows Management Console snap-in `services.msc`.

viinex 3.4 service should be restarted when configuration file or the path to configuration file is changed to reflect the changes.

Note that viinex 3.4 should not be installed on the same computer where any of previous versions of Viinex software is already installed. To install viinex 3.4 on such host, one should remove Viinex Foundation 1.4 or Viinex 2.0.

It is also possible to install viinex 3.4 in unattended mode, which can be useful for including viinex 3.4 installer into setup program of your product. For that, one may use standard command `msiexec.exe`, or use more sophisticated means involving scripting with WMI. In case of `msiexec.exe`, the following or similar syntax can be used:

```
msiexec /i Viinex-3.1.msi /quiet /log ViinexInstall.log \
    INITIAL_CONFIG="PathToConfigFile" \
    THREADS=2 \
    LOG_ROLLOVER_SIZE=16 \
    LOG_ROLLOVER_TIME=24 \
    LOG_LEVEL="INFO"
```

Both the `INITIAL_CONFIG` and `LOG_LEVEL` parameters are optional. If `INITIAL_CONFIG` is not specified or equals to special reserved value "`__KEEP__`", the registry value currently pointing to configuration file will not be overwritten. Otherwise, it is populated with specified value (which should be put in place of string "`PathToConfigFile`" in the above example).

The parameter `LOG_LEVEL` should take one of the following values: `ERROR, WARNING, INFO, DEBUG`. If `LOG_LEVEL` is not specified, the installer in unattended mode uses value `INFO` by default.

Two optional parameters `LOG_ROLLOVER_SIZE` and `LOG_ROLLOVER_TIME` specify the maximum size, in megabytes, of viinex 3.4 log file, and, respectively, the time, in hours, after which the log file should be rotated (renamed and reopened anew). When these two parameters (or one of them) are set for the installe, the latter stores specified values in Windows registry keys

```
HKEY_LOCAL_MACHINE\SOFTWARE\Viinex\Viinex30\LogRolloverSize,
HKEY_LOCAL_MACHINE\SOFTWARE\Viinex\Viinex30\LogRolloverTime.
```

That keys are used by viinex 3.4 when running in Windows service mode.

The optional parameter `THREADS` may be used to set the number of OS threads that should be used to execute viinex 3.4 code (see section 7.1.4 for details). When this parameter is set for the installer, the latter stores specified number in Windows registry key

```
HKEY_LOCAL_MACHINE\SOFTWARE\Viinex\Viinex30\Threads.
```

That key is used by viinex 3.4 when running in Windows service mode.

There are three more optional parameters which can be set for viinex 3.4 installer from command line when running `msiexec.exe`:

```
SERVICE_NAME,
SERVICE_DISPLAY_NAME,
SERVICE_DESCRIPTION.
```

These parameters allow an application that embeds viinex 3.4 to set preferrable name, display name and descrition of viinex 3.4 service. Service name is the short string to identify a service, which is used in `net start/net stop` commands. It is also displayed in the 'Name' column in the 'Services' tab of Windows Task Manager. The `SERVICE_NAME` parameter should be a short alphanumeric identifier with no whitespaces. Service display name and service description are shown in Windows Task Manager and in the Services snap-in for the Microsoft Management

Console. The three mentioned parameters are intended for better identification of viinex 3.4 service by the end-user: embedding applicaiton developer may change the default values of that properties so that service name become related to the main application name and/or branding.

### 7.1.2 Linux

For Linux, a Debian package `viinex-3.1.deb` is provided. It can be installed on a Debian-compatible operating system with command

```
dpkg -i viinex-3.1.deb
```

In contrast with Windows installer, deb-package does not contain predefined configuration files, therefor there's nothing to be configured to install viinex 3.4 on Linux.

viinex 3.4 is installed on Linux as a `systemd`-compatible service. However viinex 3.4 service does not automatically get enabled upon installation; the administrator should do so after viinex 3.4 installation with the command

```
# systemctl enable viinex.service
```

This instructs `systemd` to start viinex 3.4 service at operating system startup. If viinex 3.4 service is not enabled, it still can be started manually with the command

```
# service viinex start
```

By default, viinex 3.4 for Linux searches for its configuration file in folder `/etc/viinex.conf.d` and writes its log to the system log (using the `syslog(3)` system API). This behavior can be overridden by modifying the `ExecStart` parameter in `systemd` service description for viinex 3.4. By default, this description contains the line

```
ExecStart=/usr/sbin/viinex --log-level=INFO \
                           --syslog \
                           --config=/etc/viinex.conf.d \
                           start
```

See the subsection 7.1.3 for more details on how the command line arguments' values can be adjusted to change the log verbosity level, log file destination and the source of configuration.

The path to configuration is specified by `--config=PATH` command line argument. Its value may also point to a configuration directory rather than a single file; in that case, the split configuration logic described in section 2.13 is applied.

The log destination is set by either using the `--syslog` command line flag (which is used by default), or a `--log-file=PATH` command line parameter. There are also two optional parameters to specify the log rotation behavior, `--log-rollover-size` and `--log-rollover-time`. These parameters specify the maximum size, in megabytes, of viinex 3.4 log file, and, respectively, the time, in hours, after which the log file should be rotated (renamed and reopened anew). Note that older log files are not archived or removed by viinex 3.4 automatically. For this reason it is recommended that in production Linux-based environments the `--syslog` option is used instead, along with system-wide policies for logs rotation.

### 7.1.3 Running viinex 3.4 in foreground

In both Windows and Linux operating systems, viinex 3.4 can be run as a normal process, not as a SysV daemon or not managed by Windows NT Service Control Manager. This can be useful in the scenarios when viinex 3.4 life cycle should be controlled directly by embedding software. For instance, you may decide that starting viinex 3.4 directly using `CreateProcess` or `fork/exec` is more convenient than treating viinex 3.4 as NT service or a daemon. For that, viinex 3.4 executable accepts command-line argument `--foreground`. When this option is given, viinex 3.4 does not detach itself from the terminal in Linux and does not attempt to contact SCM in Windows. Instead, it immediately reads configuration file and tries to provide requested functionality.

Note that on Windows, when the `--foreground` option is given, viinex 3.4 does not use registry keys to determine the path to configuration file and log level. The only way to pass the the path to configuration file to viinex 3.4 is then using another command line option, `--config=`. This value may also point to a configuration directory rather than a single file; in that case, the split configuration logic described in section 2.13 is applied.

There are also additional optional command line arguments:

- `--log-level=` and `--log-file=`, which can be used to control how and where viinex 3.4 logs its runtime errors or debug information,
- `--threads=` to set the number of OS threads used by viinex 3.4 to dispatch its execution (see section 7.1.4 for details),
- `--variables=` to set the path to variables JSON file, which can be used for variables interpolation within viinex 3.4 config, as described in section 2.11.

An example command line to start viinex 3.4 would be

```
viinex --foreground --config=viinex-cfg.json \
       --log-level=INFO --log-file=viinex.log
```

Upon startup in foreground mode, viinex 3.4 is waiting for an arbitrary data on its standard input before stopping. If viinex 3.4 is ran manually in a terminal, an "ENTER" key on the keyboard may be hit to stop it. If viinex 3.4 with argument `--foreground` is started programmatically from client's software using `CreateProcess`, one should use standard input file descriptor of newly created process, which is returned in `STARTUPINFO` structure, to write an arbitrary string ending with `\r\n` to that file descriptor, when viinex 3.4 instance is required to stop.

### 7.1.4 Setting the number of OS threads

viinex 3.4 makes use of lightweight (also known as "green") threads for performing the tasks that can be done in parallel. These threads are dispatched across one or more "real" (operating system) threads, which, in turn, are scheduled to be executed on CPU cores.

viinex 3.4 allows for setting the number of OS threads employed for its execution. viinex 3.4 lets users to set that parameter as the key in Windows registry (which is done by Windows Installer if the `THREADS` property is passed to `msiexec`) or as the command-line argument `--threads=`.

By default, viinex 3.4 uses one OS thread per each CPU core for dispatching the lightweight threads. There is no sense in setting the number of OS threads above that value. However, there are cases when the number of OS threads that are used in Viinex' lightweight threads dispatching needs to be limited. Note that there may also be other ("dedicated") OS threads created by viinex 3.4, depending on what objects are specified in viinex 3.4 configuration. This is the case for video encoding and decoding components, and the video renderer object. The `--threads=` setting does not affect these dedicated threads.

The need for limiting the number of OS threads used by viinex 3.4 appears when there is a requirement to reserve the CPU cores for some other tasks running on the same server. There could also be a motivation to limit the number of threads on a Windows host with many CPU cores (like 16 and above) to save the address space in 32-bit viinex 3.4 process[1]

## 7.2 License key management

For license keys management, `viinex-lm-upgrade` utility is included in viinex 3.4 distribution, which can be used to

- enumerate attached USB dongles,
- show an information in each dongle's license time limit, the set of modules which license is written in a dongle, and the quantity of licenses,
- upgrade the USB dongle contents "remotely", without the need to send it to the licensor (in case of license prolongation or changes is the number or types of licensed objects', and
- show or update the information contained in a "emulated" license storage.

To start the license key management utility, its executable should be ran. The utility has a simple command prompt user interface. The commands to interact with `viinex-lm-upgrade` utility are described below.

The `help` displays a brief instruction on commands available in the utility.

### 7.2.1 Obtaining information on attached USB dongles

The **`enum`** command enumerates the list of keys currently attached to the computer. An example output from that command is given:

```
viinex-lm-upgrade: enum
Found 1 dongle(s)
Index   Serial
0       97502500000046ae
```

[1]Each thread requires its own stack to be allocated, which, in its turn, occupies certain amount of virtual address space. In certain workload scenarios viinex 3.4 process on the server with many CPU cores may even run out of 32-bit address space (even with modest actual memory footprint) unless the number of OS threads (and stacks) is limited.

An “index” and a serial number is shown for each USB dongle attached. For the case if more than one dongle is attached, all other commands described below accept special arguments, `-i INDEX` or `-s SERIAL`. By means of that arguments one may specify the specific dongle, which the issued command should be applied to.

### 7.2.2 Obtaining the license document from a USB dongle

The `show` command shows the content of license document stored in the USB dongle. An example output from that command is:

```
viinex-lm-upgrade: show
=====================================
Dongle serial: 9763370000000cb1
Product      : 7ae0 (Viinex20)
Time limit   : 2017-04-02 22:38:35 UTC
-------------------------------------
Features:
  ViinexCore                         1
  IpVideochannel                    16
  VideoArchive                       2
=====================================
```

### 7.2.3 Obtaining information on PC hardware

As mentioned in section 2.12, there are two kinds of license documents: those bound to a USB dongle, and those bound to a PC hardware.

If you were provided with a USB dongle to run viinex 3.4, in most cases that dongle already contains an appropriate license document in it. If a license document upgrade is required for that dongle, the viinex 3.4 licensor will be able to generate it based on USB dongle unique identifier which is reported to you by `enum` command.

However if you've requested a temporary (demo) or a permanent (production) license for viinex 3.4 to run it without a USB dongle, you'll have to provide the licensor with information about your PC hardware. Such information can be gathered using the `hwid` command to `viinex-lm-upgrade` utility. This command takes no additional arguments. An example output from that command is:

```
viinex-lm-upgrade: hwid
This PC hardware id is: kDMQptE6uNFPfFvk1hq0X14eJDeGTZKn, hypervisor not detected
```

In the above example, the hashed and base64-encoded information on PC hardware is

```
kDMQptE6uNFPfFvk1hq0X14eJDeGTZKn.
```

This string exactly should be forwarded to the licensor, along with the request for license document. In response, the viinex 3.4 licensor will provide you with a license document suitable for using in the `license` part of the configuration (see section 2.12).

### 7.2.4 Updating a license document on the USB dongle

The `update` command can be used to update the license document stored in the USB dongle. This command takes the new encrypted license document as its mandatory argument. An example output from the `update` command is:

```
viinex-lm-upgrade: update POR2n3jaCzZW/8FM12znbke3y0gBsXn2/Kzus4xKPYN3QNdtgIJKWK
xMjCrObpsSG5Oev0ndT3pw79FuwwIPkAbaGZSh/lP6GHQA9eaJHUagtU+C2/pYeU3ncPgjdc6B4p3CE4
ooaA+8kBSo4ACInzzv2noefdCFrZJUD1tKc3TYuqHpEi+xkcFft7I/34qV03JErgSl+y0GQBqAHTFtDw
sPF3VDIeRCBub3KeTm1XC6JivwrLQ0mp3QacSdSKhPqmYUGuxzZbL/aoOyiFNJVU4Cah6YjMC/RvMpNi
LAfOM=
License document updated; firmware responded: 97502500000046ae
```

The message in the last line of the output means that license document was updated successfully. When this happens, the firmware built into the USB dongle replies with dongle serial number; this is an indicator of the fact that the firmware has ran without errors. In case of errors the reply message is different:

```
viinex-lm-upgrade: update 9RIkOti...pKeZw=
Error: license document update failed: 0651
```

Such message is produced if an attempt is made to update the dongle which serial number differs from the one for which the license document was issued. Error code (0651 in above example) can be reported to Viinex support team to diagnose the actual reason of license document upgrade failure.

### 7.2.5 Working with an "emulated" license storage

The `viinex-lm-upgrade` utility is capable of working with so called "emulated" license storage. An "emulated" license storage is a fixed place in the operating system where a license document can be stored. To be precise, on Windows this is a registry key

`HKLM\SOFTWARE\Viinex\Viinex30\License.`

On Linux, this is a binary file `/var/lib/viinex/license`.

The purpose of the concept of "emulated" license storage is to mimick the behaviour of a Senselock dongle holding Viinex license document, with the absence of the Senselock dongle itself. A software-bound license document (i.e. the document bound to a hardware ID or to a MAC address) may be written to an emulated storage by means of command `update`, and it can be read from that storage by means of command `show`, just as described in sections 7.2.4 and 7.2.2. The differences from `update` and `show` commands working with USB dongles are that

1. An "emulated" storage is only used when no Senselock dongles are attached to the host. There are no command line switches to force the use of the "emulated" storage when a Senselock dongle is attached. An "emulated" storage is chosen implicitly. The same is true for viinex 3.4 itself: an attempt is made to read the license document from "emulated" storage if and only if there is no USB dongles attached (and also if there is no `license` parameter specified in viinex 3.4 configuration. Respective policies are described in section 2.12).

2. The `update` and `show` commands working with an emulated storage only accept the software-bound license documents, while that commands for USB dongles only accept license documents for USB dongles.

3. While `viinex-lm-upgrade` verifies a license document that is being written to the "emulated" license storage by means of `update` command, – it does not make any checks whether this specific license document is suitable for the current host. This behaviour is different from the `update` command for USB dongles, where it is impossible to write a license document on the dongle for which this document was not intended. So some additional care needs to be taken about this with software-bound license documents and "emulated" storage.

4. Unlike the USB-oriented counterpart, the `update` command for an "emulated" storage requires an elevated privileges on Windows. On Linux there is no difference because `viinex-lm-upgrade` needs root privileges to access a USB device, and it likewise needs root privileges to create or modify a file in `/var/lib/viinex` directory.

> **NB!** In order to `update` a license document in the "emulated" storage, `viinex-lm-upgrade` utility needs to be run with elevated privileges (Run as administrator) on Windows.

To sum up, an "emulated" license storage provides the mechanism to specify a software-bound license document for viinex 3.4 without the need to modify viinex 3.4 configuration, which may be convenient in some use cases. An "emulated" license storage gets checked by viinex 3.4 runtime in the last place, if no license documents can be found in conventional places (i.e. viinex 3.4 configuration and attached Senselock USB dongles).

### 7.2.6 Batch mode

All commands available in `viinex-lm-upgrade` utility can be issued not only in the interactive mode, but in the batch mode as well. For this, full text of the command should be appended to the name of `viinex-lm-upgrade` utility as its command line arguments. For example:

```
# ./viinex-lm-upgrade show -s 9763370000000cb1
=====================================
Dongle serial: 9763370000000cb1
Product      : 7ae0 (Viinex20)
Time limit   : 2017-04-02 22:38:35 UTC
-------------------------------------
Features:
  ViinexCore                         1
  IpVideochannel                    16
  VideoArchive                       2
=====================================
```

# References


[1] ISO/IEC 14496-12:2015. Information technology – Coding of audio-visual objects – Part 12: ISO base media file format http://www.iso.org/iso/catalogue_detail.htm?csnumber=68960

[2] ISO/IEC 13818-1:2015. Information technology – Generic coding of moving pictures and associated audio information – Part 1: Systems http://www.iso.org/iso/ru/home/store/catalogue_tc/catalogue_detail.htm?csnumber=67331

[3] R. Pantos, Ed., et al. HTTP Live Streaming (RFC draft) https://datatracker.ietf.org/doc/draft-pantos-http-live-streaming/

[4] H. Schulzrinne et al. Real Time Streaming Protocol (RTSP) https://www.ietf.org/rfc/rfc2326.txt

[5] H. Schulzrinne at al. RTP: A Transport Protocol for Real-Time Applications https://www.ietf.org/rfc/rfc3550.txt

[6] Y.-K. Wang et al. RTP Payload Format for H.264 Video https://tools.ietf.org/rfc/rfc6184.txt

[7] H. Krawczyk, M. Bellare, R. Canetti. HMAC: Keyed-Hashing for Message Authentication https://tools.ietf.org/rfc/rfc2104.txt

[8] ONVIF Core Specification. Version 2.2, May 2012 https://www.onvif.org/specs/core/ONVIF-Core-Specification-v220.pdf

[9] ONVIF Media Service Specification. Version 2.4, August 2013 https://www.onvif.org/specs/srv/media/ONVIF-Media-Service-Spec-v240.pdf

[10] ONVIF Imaging Service Specification. Version 2.2.1, December 2012 https://www.onvif.org/specs/srv/img/ONVIF-Imaging-Service-Spec-v221.pdf

[11] ONVIF Device IO Service Specification. Version 2.2, May 2012 https://www.onvif.org/specs/srv/io/ONVIF-DeviceIo-Service-Spec-v220.pdf

[12] ONVIF PTZ Service Specification. Version 2.2.1, December 2012 https://www.onvif.org/specs/srv/ptz/ONVIF-PTZ-Service-Spec-v221.pdf

[13] J. Franks et al. HTTP Authentication: Basic and Digest Access Authentication https://tools.ietf.org/html/rfc2617

[14] YUV Video Subtypes. https://msdn.microsoft.com/en-us/library/windows/desktop/dd391027%28v=vs.85%29.aspx

[15] I. Fette, A. Melnikov. The WebSocket Protocol https://tools.ietf.org/html/rfc6455

[16] M. Baugher et al. The Secure Real-time Transport Protocol (SRTP) https://tools.ietf.org/html/rfc3711

[17] E. Rescorla, N. Modadugu. Datagram Transport Layer Security Version 1.2 https://tools.ietf.org/html/rfc6347

[18] D. McGrew, E. Rescorla. Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP) https://tools.ietf.org/html/rfc5764

[19] J. Rosenberg et al. Session Traversal Utilities for NAT (STUN) https://tools.ietf.org/html/rfc5389

[20] A. Keranen, C. Holmberg, J. Rosenberg. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal https://tools.ietf.org/html/rfc8445

[21] M. Handley, V. Jacobson, C. Perkins. SDP: Session Description Protocol https://tools.ietf.org/html/rfc4566

[22] S. Nandakumar, C. Jennings. Annotated Example SDP for WebRTC (draft) https://tools.ietf.org/html/draft-ietf-rtcweb-sdp-11

[23] ECMA International. ECMAScript Language Specification (ECMA-262, 5.1 Edition) http://www.ecma-international.org/ecma-262/5.1/ECMA-262.pdf

[24] Viinex 2.0 demo user interface implementation source code. https://github.com/viinex/viinex-demo-ui/blob/5026c9c7b97ccd50b95007044742fe453bbf20d8/src/app/login.service.ts#L104

[25] M. Jones, J. Bradley, N. Sakimura. JSON Web Token (JWT) https://datatracker.ietf.org/doc/html/rfc7519

[26] T. Oberstein, A. Goedde. The Web Application Messaging Protocol https://wamp-proto.org/_static/gen/wamp_latest_ietf.html

[27] WebRTC 1.0: Real-Time Communication Between Browsers. W3C Editor's Draft 08 November 2021 https://w3c.github.io/webrtc-pc/#rtciceserver-dictionary

[28] Cryptosign Authentication. Crossbar.io Application Router https://crossbar.io/docs/Cryptosign-Authentication/

[29] HTMLVideoElement.requestVideoFrameCallback(). W3C Community Group Draft Report https://wicg.github.io/video-rvfc/

[30] Prometheus Exposition Formats https://github.com/prometheus/docs/blob/main/content/docs/instrumenting/exposition_formats.md#text-based-format