Viinex 3.0

User's Guide

2020

Contents

This document is a reference manual to help application developers in configuring and using Viinex 3.0.

If you have technical questions on Viinex 3.0, please contact the support team at Viinex helpdesk: https://viinex.atlassian.net/servicedesk/customer/portal/1/

Compiled on September 14, 2020.

1 General information

Viinex 3.0 is a software development kit (SDK) for adding video surveillance and video management features to customer's application. Viinex 3.0 implements functionality for acquiring a video data from external devices (IP cameras and encoders), storing video in the video archive, re-streaming video to clients in live mode as well as on demand. It also allows your application to decide when video from external sources should be recorded, and takes care of such low-level aspects of video management as video buffering for pre-recording, precise handling of groups of pictures and key frames in a video stream for exact positioning within video archive, accurate accounting for presentation timestamps, etc.

In its interaction with applications, particularly in video data interchange, Viinex 3.0 sticks to ISO-standardized media formats. When dealing with H.264 video codec, Viinex 3.0 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 3.0 implements HLS specification [3], making video playback possible on most of popular browsers, including Microsoft Edge and Apple Safari on iOS. Internally, Viinex 3.0 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 3.0 was installed and brought to another standard PC with no additional software).

Viinex 3.0 is inherently embeddable, and does not inevitably bring its own end-user interface to the product where it is used. Viinex 3.0 is completely separated from customer's application address space; there's no need for linking your code with Viinex 3.0 client libraries (in fact for Viinex 3.0 there are no such client libraries at all). All interaction with Viinex 3.0 is performed via HTTP REST-like programming interface, which can be reached from wide range of programming languages, from C/C++ to shell scripts.

While being a simple local service/daemon, Viinex 3.0 is capable of participating in a distributed system for video management. Out of the box, Viinex 3.0 contains built-in replication modules, which allow for eventual synchronization of video archives between Viinex 3.0 instances.

2 Configuration

Viinex 3.0 is started with one or more configuration files of simple JSON format, which sections' semantics is described below. The configuration for Viinex 3.0 is a JSON document (or several documents, see section 2.6) containing three optional keys: `objects`, `links` and `license`. 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"
}
```

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
        },
        {
            "type": "alpr",
            "name": "alpr0",
            "country": "DEU",
            "datapath": "/home/viinex/share/lib/vnxlpr/",
            "workers": 1
        }
    ],
    "links":
    [
        [["cam1", "cam2"], "stor0"],
        ["web0", "alpr0"],
        ["web0", "stor0"]
    ],
    "license": "rqZ821uWtcsxz....fYZE76ByAgmCZO"
}
```

2.1 Objects

The `objects` section is a JSON array for elementary functionality units ("objects", or module instances) to be run when Viinex 3.0 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 3.0 (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 3.0 component with some information coming from the application which uses Viinex 3.0. 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.

2.1.1 RTSP video source

Viinex 3.0 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 3.0 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"],
  "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 3.0 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 2.4.1. performed. If `transport` parameter is not specified, Viinex 3.0 RTSP client defaults to `"transport": ["udp", "tcp"]` which effectively enables UDP unicast and TCP (in that order of preference) but disables UDP multicast.

The optional `dynamic` property, when set to `true`, instructs Viinex 3.0 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[1] (while the video recording of this RTSP video source is requested),
- a Web server[2] (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 3.0 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 3.0 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 3.0 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.

[1] 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.

[2] 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 3.0 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.

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 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 3.0 objects please refer to section 3.19). 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[3]

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,
```

[3]This is because Viinex 3.0 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 3.0 cannot infer how many frames were lost and does not account for such packets and frames. Moreover, if Viinex 3.0 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%.

```
  "origin": {
    "type": "rtsp",
    "name": STRING
  },
  "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.

2.1.2 ONVIF device

Viinex 3.0 supports ONVIF specification for acquiring H264 video streams and events from ONVIF devices. Configuration object for ONVIF device in Viinex 3.0 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`[4] which is a JSON array and may hold from zero to three elemetns — `"video"`, `"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). 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 and events, like this:

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

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 3.0 via WebSocket interface described in section 3.19. 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`.

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

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 2.4.1.

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 3.0 automatically selects the first[5] 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", "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 2.1.9.

The implementation of event source interface of "ONVIF device" module in Viinex 3.0 does not require any settings. Viinex 3.0 automatically creates pull-point subscription and begins to acquire all events produced by ONVIF device, – right after an instance of the module is started.

2.1.3 H264 video source plugin

In order to provide applicaitons with capability to make arbitrary video streams available as live video sources in Viinex 3.0, the latter supports the functionality of so-called `h264sourceplugin`.

[5]The order is actually defined by the device. Viinex 3.0 takes the first appropriate profile that is described in the result to `GetProfiles` ONVIF call [9].

The idea behind that functionality is that an application can provide a shared library that implements certain simple API, which is described in section 5.3, calls the specified factory method in that library, which should return a specific implementation of a video source[6]. That video source is then used by Viinex 3.0 just like any other video source.

The configuration for the H264 live video source plugin should look as follows:

```
{
    "type": "h264sourceplugin",
    "name": STRING,
    "dynamic": BOOLEAN,
    "library": STRING,
    "factory": STRING,
    "init": JSON
}
```

where property `type` should take the value of `"h264sourceplugin"`; value of property `name` identifies the video source in Viinex 3.0 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 3.0 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 3.0 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 3.0 and, once found, is called, in order to instantiate the pluggable video source.

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 3.0 prior to the calling of `factory` function inside of the `library`. The stringified `init` value is actually passed by Viinex 3.0 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 3.0, and therefore it is present with every Viinex 3.0 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": "h264sourceplugin",
     "name" : "cam2",
     "library" : "vnxvideo.dll",
     "factory" : "create_media_file_live_source",
     "init" : {
```

[6]To be precise, the `vnxvideo_h264_source_t` type and the C++ interface `IH264VideoSource` represent a H264 video source co-located with event source, so that a plugin can produce events synchronized with a video stream. For more details refer to section 5.3.

```
        "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_media_file_live_source` (compare to https://github.com/viinex/vnxvideo/blob/9acd9a06/src/FileVideoSource.cpp#L334). 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.

2.1.4 Raw video source

Viinex 3.0 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 3.0 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",
        "address": STRING |
        "index": INT,
        "mode": OBJECT
    }
```

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

The `address` property should be a string value of address (absolute path) of the device to connect to. On Windows, 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, 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 3.0 API documented in section 3.3.7.

Alternatively, instead of `address`, an `index` may be specified. `index` is an integer interpreted as an ordinal zero-based number of the device that Viinex 3.0 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 3.0.

The `mode` property of `capture` structure should have the form described in section 2.4.3. `mode` property is optional. If not specified, Viinex 3.0 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 3.0 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 3.0 is "I420". Viinex 3.0 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.

Encoder

Video encoder is always created and associated with raw video source by Viinex 3.0: 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 2.4.4.

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 3.0 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 3.0. 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 2.1.9, 3.19 for more details.

Overlay

The `overlay` subsection is optional. If present, it enables the overlay functionality for the raw video source in Viinex 3.0. The syntax and meaning of the properties under the `overlay` configuration section are given in paragraph 2.4.5 of this document.

2.1.5 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 3.0 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,
  "dynamic": true,
  "share": true,
  "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]
      }
    ],
    "encoder": {
      "type": "cpu",
      "quality": "small_size",
      "profile": "high",
      "preset": "ultrafast"
    }
  }.
```

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` 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 2.4.8.

The `overlay` parameter is optional and defines the overlays rendered over the resulting video. The syntax and semantics of this section is described in 2.4.5.

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 5.2 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 3.0 may be run with an "evevated" privileges ("Run as administrator"). When running as a service, Viinex 3.0 uses the SYSTEM account by default, which already has the required privileges to create a shared memory segment.

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 2.4.4.

There is also an optional boolean parameter `dynamic` in the configuration of renderer, 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. Note that `dynamic` property set to `true` only affects the encoded video stream. The rendering of a raw stream is performed permanently. The default value of `dynamic` property is `false`.

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.

Video renderer implements a number of interfaces for interaction with other components implemented in Viinex 3.0. 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 2.2.

2.1.6 Stream switch

Stream switch provides an efficient way to multiplex several video sources into one, giving an application using Viinex 3.0 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 – producting 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.

2.1.7 Video archive

Viinex 3.0 implements the functionality for writing, storing and accessing H.264 video data acquired from external video sources.

Configuration object for video archive in Viinex 3.0 is denoted by object type `storage`. Such configuration object should contain three mandatory fields: `folder`, `filesize` and `limits`. An example for video archive configuration is given below:

```
{
  "type": "storage",
  "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.

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 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 3.0 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 3.0 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 3.0 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 3.0 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.

2.1.8 Recording controller

Viinex 3.0 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 2.1.9, 3.6.

An instance of recording contoller is denoted in Viinex 3.0 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.

2.1.9 Rules

As an alternative for an external management of video recording via HTTP remote procedure calls to the recording controller, Viinex 3.0 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 3.0:

```
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[7]. `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

[7] 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 3.0 configuration.

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).

2.1.10 Replication source

Viinex 3.0 implements replication functionality split into two parts: replication source and replication sink. Replication source is responsible for sending video data from Viinex 3.0 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 2.1.11 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 3.0 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.

2.1.11 Replication sink

Replication sink is Viinex 3.0 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 3.0, 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 2.1.13).

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 3.0 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 2.1.21 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 3.0 instances. Doing so is a normal use case; corresponding configuration example is provided in stock configuration files coming with Viinex 3.0 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 3.0 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 3.0 configuration, but it should not exceed the number allowed in the license document, otherwise Viinex 3.0 won't be able to start such configuration cluster.

Since the destination for video data replication (the channel within the video archive) is governed 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.

2.1.12 Modbus GPIO-related event source

Besides integration with ONVIF-compatible devices and registering events originating from such devices, as described in section 2.1.2, Viinex 3.0 supports obtaining of GPIO-related events from devices that support Modbus TCP protocol (typically a GPIO controller with an Ethernet interface).

Respective object in Viinex 3.0 has the implementation type `modbus`, and its configuration should have the form of:

```
{
    "type": "modbus",
    "name": STRING,
    "protocol": "modbus" | "vkmodules1",
    "host": STRING,
    "port": INT,
    "inputs": INT
}
```

There are two mandatory parameters, that is `host` and `inputs`, and two optional parameters, `protocol` and `port`.

The optional parameter `protocol` specifies the wire protocol that should be used to talk to the controller device. This parameter should have either the value `modbus`, which stands for Modbus TCP, or a vendor-specific value for communicating with modbus-incompatible hardware[8] supported by Viinex 3.0. If this parameter is omitted from the configuration, the value `modbus` is assumed.

The parameter `host` specifies the IP address of the device implementing GPIO functionality. It can be accompanied by the parameter `port` to specify the TCP port to use for communication. If the `port` is omitted, the default value for respective protocol is assumed, which is TCP port number 502 for Modbus[9].

The parameter `inputs` specifies how many digital inputs should be monitored. For small number of inputs (8 and below) it is safe to set this value to the total number of digital inputs that are supported by the controller. If the total supported number of inputs is big, the number of monitored inputs (and, consequently, the quantity of generated events) can be limited with this parameter. Note, however, that the digital inputs in Modbus are addressed by an index value in range [0,255], and there is currently no way how the lower bound of the range of monitored inputs can be set in Viinex 3.0. That is, no matter to which value K the parameter `inputs` is set, – Viinex 3.0 would monitor the digital input with indices $[0 \ldots K]$ (always starting from 0).

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 noticed, the respective event containing the new state is produced.

The `modbus` object currently does not support controlling actuators via digital outputs, therefore it does not expose any HTTP API methods except the one for acquiring the current

[8] As per December 2018, the only value alternative to `modbus` supported by Viinex 3.0 is `vkmodules1`, which stands for "VKModule Socket-1" controller, for more information see http://vkmodule.com.ua/Ethernet/Ethernet2.html.

[9] and port number 9761 for the "VKModule Socket-1" controller.

state of digital inputs. This is performed via the abstract `Stateful` interface described in section 3.18.1. This method implemented by the `modbus` object returns a JSON array of values representing the current states of digital inputs.

This object acts as an event source, implementing a semantical 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`).

2.1.13 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 3.0 supports integration with a number of video management systems from third-party vendors.

In Viinex 3.0 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 3.0 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 3.0 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 2.3.

The second part of third-party VMS integration to Viinex 3.0 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 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. For instance, there could be an additional property `stream`, identifying a substream from the video source:

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

It depends on a particular third-party VMS and its integration into Viinex 3.0 whether such additional properties are supported. Please refer to section 2.3 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 `trassir`, see section 2.3):

```
"links": [ ...
  ["trassir1", ["chan1", "chan2", ...]],
... ]
```

The absence of such connection for any of `vmschan` objects would result in an error during Viinex 3.0 startup.

The objects of type `vmschan` can also be used in the links with other Viinex 3.0 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.

2.1.14 Vehicle license plate recognizer

License plates text recognition engine for still images

Viinex 3.0 implements the functionality for vehicle license plate recognition on still images as well as on a video. License plate recognition on still images is implemented in an object that can be used to either perform such recognition when images come from external source via HTTP API, or when they come from a video source, or both.

Configuration object for an instance of the LPR is denoted by the object type `alpr`. An example of such configuration object is given below:

```
{
  "type": "alpr",
  "name": "alpr0",
  "mode": "image",
  "datapath": "share/vnxlpr/",
  "country": "DEU",
  "workers": 4
}
```

Configuration parameters `mode`, `datapath` and `workers` are common for analytics engines and are described in section 2.4.6. The rest of parameters are specific to license plate recognition engine, and their description is given below.

The mandatory field `country` should contain a 3-letter ISO code for country which has issued the license plates to be recognized. Please contact Viinex 3.0 support team to get most recent information on what countries are supported in vehicle license plate recognition engine.

The `transliterate` field is optional and provides the possibility to transliterate recognition result to a different character set. For instance, this might be useful if the recognition result is compared against a database (a whitelist or a blacklist), where the license plate numbers are stored written in cyrillic letters. If `"transliterate": "cyrillic"` is specified, then the latin letters in recognition result are automatically replaced with cyrillic letters of similar shape, like "H (latin h capital)" vs "H (cyrillic н capital)" and so on. If `transliterate` option is not set, or its specified value is "latin", then the recognition results are produced with latin letters in them.

License plates text recognition for video

To perform license plates text recognition on a data from a video source, the Viinex 3.0 object of type `alprvideo` is used.

The `alprvideo` object is essentially a controller which takes the video stream from a video source, exposes a simple API in a web server to receive a request for recognition, and when such request is received via HTTP API (see section 3.15.2), it performs license plate text recognition on a buffered sequence of video frames using an instance of `alpr` object which is functioning with `mode` parameter set to ''`video`'' or ''`any`''.

Note that this is necessary for `alprvideo` object to function that it is linked (in the `links` section of configuration document) with an instance of `alpr` object, serving as the license plates text recognition engine, and a video source to take the data for analysis from.

Configuration section for `alprvideo` object should have the form similar to the following example:

```
{
    "type": "alprvideo",
    "name": "alprraw0",
    "preprocess": 1.5,
    "postprocess": 1.0,
    "skip": "non_idr",
    "confidence": 40
}
```

The complete list of configuration parameters and their detailed description is given in section 2.4.7.

2.1.15 Face detection

Face detection engine for still images

Viinex 3.0 implements the functionality for face detection on still images as well as on a video. Face detection on still images is implemented in an object that can be used to either perform such detection when images come from external source via HTTP API, or when they come from a video source, or both.

Configuration object for an instance of the face detector is denoted by the object type `facedet`. An example of such configuration object is given below:

```
{
    "type": "facedet",
    "name": "facedet0",
    "mode": "any",
    "datapath": "share/facedet/",
    "workers": 2,
    "min_size": 0.05,
    "max_size": 0.9
}
```

Configuration parameters `mode`, `datapath` and `workers` are common for analytics engines and are described in section 2.4.6.

The `min_size` and `max_size` parameters define the scales at which the input image is searched for the faces. Given the above example, the face detection engine is set up to search for the faces in the range from 5% to 90% of image size. For instance, if the input image is of size 1280×720, the sizes of faces to search for wolud be from 36 to 648 pixels.

The important thing about this range is that it heavily influences the usage of CPU resources by the face detection engine. This is because the search for faces at different scales is performed by means of a number of sequential steps each consisting of the re-scaling of the input image, and performing the search of a faces of one fixed size at given rescaled image. The steps performed closer to the `min_size` boundary are more computation-intensive than those closer to the `max_size` boundary (that is — the detection of smaller faces requires more CPU cycles than the detection of larger ones). It is therefore recommended that in production environment the range [`min_size`, `max_size`] is made as narrow as possible, if the range of scales at which the faces should be detected is known in advance.

Face detection for video

To perform face detection on the data from a video source, the Viinex 3.0 object of type `facedetvideo` is used.

The `facedetvideo` object is the object controller which takes the video stream from a video source, exposes a simple API in a web server to receive a request for face detection, and when such request is received via HTTP API (see section 3.17.2), it performs face detection on a buffered sequence of video frames using an instance of `facedet` object which is functioning with `mode` parameter set to ''`video`'' or ''`any`''.

Note that this is necessary for `facedetvideo` object to function that it is linked (in the `links` section of configuration document) with an instance of `facedet` object, serving as the face detection engine, and a video source to take the data for analysis from.

Configuration section for `facedetvideo` object should have the form similar to the following example:

```
{
    "type": "facedetvideo",
    "name": "facedetraw0",
    "preprocess": 0,
    "postprocess": 10,
    "skip": "while_busy",
    "confidence": 80
}
```

The complete list of configuration parameters and their detailed description is given in section 2.4.7.

2.1.16 Railcar identification number recognition

Viinex 3.0 has a built-in support for coordination and controlling of video analytics engine

processes' in order to perform recognition of railcar identification number[10]. Respective object type is `ridrcons`. An example for `ridrcons` configuration is given below:

```
{
    "type": "ridrcons",
    "name": "ridrcons1",
    "datapath": "share/vnxrlw",
    "channels": {
        "cam1": {
            "min_width": 0.05,
            "roi": [0,0,1,1]
        },
        "cam2": {
            "min_width": 0.05,
            "roi": [0,0,1,1]
        }
    }
}
```

The `datapath` parameter should contain the path to analytics engine data files, which should be `share/vnxrlw` on Windows. The `channels` section of configuration should contain an associative array mapping the name of video sources connected to the `ridrcons` instance to a JSON object describing the geometry of image on each video source. In particular, the `min_width` parameter specifies the minimum width of railcar identification number (size relative to the width of the image), while the `roi` specifies the region of interest – a rectangle on the image where the analytics should be performed (in form of consecutive 4 floating point numbers, – left, top, right and bottom of the rectangle, measured relative to image width and image height).

The `ridrcons` object should be linked with video sources in the `links` section of configuration. It should also be linked with other objects, like web server or controlling script, in order to implement the necessary behaviour. Typically the `ridrcons` is used in conjunction with a controlling script which, in its turn, may process specific events from a GPIO controller, or a user calls from HTTP API, and respectively control the `ridrcons` object using the `Updateable` interface.

2.1.17 Script

In order to provide flexibility for using Viinex 3.0 in various scenarios, the latter has built-in support for scripting.

That support is implemented just like all other objects' implementations in Viinex 3.0. Namely, an object of type `script` is introduced. Objects of that type represent an instance of JavaScript engine[11]. 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 3.0:

[10] Only relevant for 1520 mm gauge railways.

[11] To be precise, Viinex 3.0 uses the Duktape engine https://duktape.org/ which implements ECMAScript 5 [23] as per November 2018.

- maintain internal state according to a custom logic, and expose a part of that state via HTTP API (see section 3.18.1);
- accept simple requests to update the internal state and reply to such requests to the parties who initiate them (see section 3.18.2);
- 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 3.0), 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 3.0 startup, otherwise the `script` object reports configuration error. The files mentioned in this parameter are executed sequentially, in the order they are mentioned.

The parameter `inline` may contain an inline JavaScript code, – that is, some part of code can be included directly in Viinex 3.0 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 3.0 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 3.0 objects (and they might be not created by the time). None of API specific to Viinex 3.0 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.

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 3.0

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 3.0 objects are linked. Basically, this is an entry point, when the instance of script can complete its initialization, having the access to Viinex 3.0 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 3.18.2 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 3.0 configuration clusters, as described in section 4.2.6. 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 3.0 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 3.0 objects that are linked with this instance of `script` in the `links` section of the configuration. The Viinex 3.0 objects that are not linked with the script are inaccessible for it.

Each instance of the `script` object in Viinex 3.0 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 3.18).

For more information on scripting in Viinex 3.0 please refer to chapter 4.

2.1.18 External process

Overview

Besides the scripting capability, Viinex 3.0 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 3.0 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 3.0 configuration.

There are two mechanisms provided for communication between Viinex 3.0 and an external process, which serve for two different purposes:

- Events interchange. An external process can obtain events from Viinex 3.0, and that process can also generate events and pass them to other objects in Viinex 3.0. For that, the simple mechanism of standard input/output is used. Viinex 3.0 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 3.0 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 3.0 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 3.0 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 3.0 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 5.2.

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 3.0 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 3.0 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 3.0 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 3.0 instance; wherein the variables provided in the `env` array override those from inherited environment.

The lifecycle of the process is managed by Viinex 3.0. This means that the process is started by Viinex 3.0, and the latter watches for the status of the process. If the process stops, Viinex 3.0 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 3.0 tries to shut down its child processes gracefully. In order to do that, Viinex 3.0 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 3.0 would forcefully terminate such process. The parameter `timeout` defines

the time interval, in seconds, that Viinex 3.0 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 3.0. 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 3.0 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 3.0 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 3.0 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 3.0 configuration.

Another IPC channel provided between Viinex 3.0 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 3.0 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 `INFO`.

As stated in the overview, implementations are free to use the functionality for obtaining raw live video streams from the instance of Viinex 3.0 they are supervised by, using the API briefly described in chapter 5.

2.1.19 RTSP server

Viinex 3.0 has a built-inf RTSP server for streaming video to remote clients, including other instances of Viinex 3.0.

Configuration object for RTSP server in Viinex 3.0 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,
    "auth": {
        "require": true,
        "realm": "ViinexAuth",
        "htdigest": "C:/htdigest.txt",
        "accounts": [
            {
                "type": "password",
                "login": "admin",
                "digest": "510070c93040af4bef5e6d311d26af74"
            }
        ]
    },
    "transport": ["udp","tcp","mcast"],
    "mcast_base": ["224.0.0.70", 20000]
}
```

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 defines the credentials for accessing RTSP server by clients. Its syntax and semantics is described in section 2.4.2. Viinex 3.0 only supports digest authentication for RTSP clients [13]. If `auth` section is missing from configuration of the RTSP server instance, or the value `auth.require` is set to `false`, – the instance of RTSP server does not require authentication from connecting clients.

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 2.4.1. `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 3.0 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 3.0 RTSP server instance. Viinex 3.0 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 3.0 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 3.0 is the multicast group address and base port number.

To specify which video sources and/or video archives should be accessible via Viinex 3.0 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 3.0 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.

2.1.20 WebRTC server

Viinex 3.0 is able to restream the live video 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 live video sources in Viinex 3.0, 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.

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

```
{
    "type": "webrtc",
    "name": "webrtc0",
    "stun": [["stun.l.google.com", 19302]],
    "stunsrv": 3478,
    "key": "etc/ssl/private/sample-privkey.pem",
    "certificate": "etc/ssl/private/sample-certificate.pem"
}
```

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[12] are installed when deploying Viinex 3.0 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
```

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

The `stun` parameter may contain a JSON array of pairs (which is, in turn, encoded as a JSON array of 2 elements) of a string – a hostname or IP address – and an integer – a port number of a STUN server. There can be multiple STUN server specified in the configuration, and there can be zero number of STUN servers as well. In the latter case, the WebRTC server will only be reachable by remote clients if their traffic can be directly routed to the server where the instance of Viinex is running (i.e. if the client and the server are on the same network, or if the server has a "white" IP address on the Internet). If the client and the Viinex 3.0 server are both behind the NAT, the `stun` parameter should be specified.[13]

Viinex 3.0 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 3.0, 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 3.0 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.

In order to publish live video sources using the WebRTC server implemented in Viinex 3.0, 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.

Currently (as per September 2019) the implementation of WebRTC in Viinex 3.0 has the following limitations:

- Only publishing of live video sources is supported.
- There can be only one video source at a time, published in a single WebRTC session. However an existing WebRTC session can be switched between video sources, without the need to establish a new session.
- Viinex 3.0 does not support (initiate) routing media data via TURN servers.

2.1.21 Web server

Viinex 3.0 has a built-in web server implemented for making media data accessible, and to expose API for applications using Viinex 3.0.

Configuration object for web server in Viinex 3.0 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",
```

[13] There is a list of publicly available STUN servers on the Internet. In particular, Google has its STUN servers available, in the exable above the DNS name `stun.l.google.com` resolves to a server matching the geolocation. The traffic generated by Viinex 3.0 to such STUN server is neglectable – it's only a few UDP datagrams per one WebRTC session. However, Google does not provide support for using their servers by thirdparty applications. Therefore it is recommended that an instance of STUN server is deployed for production use of WebRTC feature by the application developer who uses Viinex 3.0 video management SDK.

```
  "name": "web0",
  "port": 8880,
  "staticpath": "/usr/share/viinex/html",
  "static": [
    ["images","/usr/share/images"],
    ["logs","/var/log"]
  ],
  "auth": {
    "require": true,
    "realm": "ViinexAuth",
    "htdigest": "C:/htdigest.txt",
    "accounts": [
      {
        "type": "password",
        "login": "admin",
        "digest": "510070c93040af4bef5e6d311d26af74"
      },
      ...
      {
        "type": "apikey",
        "key": "remoteagent1",
        "secret": "foobarsecret1"
      },
      ...
    ]
  },
  "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"]
  },
  "cors": "*",
  "hls": {
    "fragments": 3,
    "duration": 5
  },
  "clusters": true
}
```

The field `port` defines a TCP port number which Viinex 3.0 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 3.0 static web content is deployed. The contents of specified folder is served by Viinex 3.0 web server starting at root URL.

The `auth` determines how the clients should authenticate themselves when accessing the web server. The syntax and semantics of this section is covered in paragraph 2.4.2. If `auth` section is omitted, or the value `auth.require` is set to `false`, the instance of web server requires no authentication from its clients, this permitting the anonymous access.

Note that the value of `auth.require` parameter in webserver configuration only affects the public part of HTTP API. There is also a private part of API in Viinex 3.0, which is not covered by this documentation. This private API is used in interaction between replication sink and

replication source. Private API always requires authentication (that is, `auth.require` value of corresponding web server configuration does not affect private API protection). Therefore, even if the value `auth.require` is set to `false`, in the web server which is used to expose the Replication Source, an `auth` section should be configured to match the credentials (API keys and secrets) set at corresponding remote instances of Replication Sinks. Otherwise, replication sinks will not be able to authenticate against replication source and upload the data.

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.

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 3.0 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 3.0 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 3.20 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`.

2.1.22 Publisher for objects in configuration clusters

Publishing of objects created in the main static configuration of Viinex 3.0 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 3.20.6. 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 3.0 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 3.20.

For the purpose of section 2.2, 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.

2.2 Links

The `links` section contains an array of JSON arrays, each describing one or more functional connections that should be established between Viinex 3.0 units at runtime. 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.

The basic form of a link is a pair of names of Viinex 3.0 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 3.0 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.

The exact semantics of a basic link created between two Viinex 3.0 objects is automatically inferred from that object's types. Not every two objects can be linked together: it depends on objects' types whether the link between such objects is meaningful in Viinex 3.0. 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 3.0).

An attempt to create a link between two functinally incompatible objects causes an error at Viinex 3.0 startup. Concrete descriptions of allowed links are given in section 2.2.

Section `links` of the configuration documents defines how components are connected to work together. To define a functional connection (link) between Viinex 3.0 objects, one has to specify each pair of objects that should interact. Each object implements one or several *interfaces* – atomic units of functionality. Some examples for these interfaces are:

- video source,
- snapshot source,
- overlay control,
- layout control,
- PTZ control,
- video storage,
- recording controller,
- event source.

Objects' interaction type is inferred automatically from types of objects being connected, or, more precisely, from the interfaces implemented and exposed by objects being linked. That is, for instance, in connection of ONVIF device and a video storage, and in connection of RTSP video source and a video storage, significant is that both ONVIF device and RTSP source implement the video source interface. If an object implements several interfaces, and such object is encountered in `links` section, – Viinex 3.0 attempts to link all interfaces provided by that object with all interfaces provided by second object, expecting that at least one pair of interfaces can be linked successfully. For instance, "ONVIF device" object implements several interfaces, including video source and snapshot source. When linked to a web server, which is capable of publishing both video source and a snapshot source, – both interfaces of ONVIF device are used, that is – there may be one link specified between ONVIF device and a web server, but essentially two links are created, one for publishing a video source, and one for snapshot source.

Not every two types of objects can interact, however. If an "illegal" connection (a pair of functionally incompatible objects forming a link) is encountered in configuration document, an error is reported. The objects are said to be functionally incompatible if none of their interfaces are compatible. That is, at least one interface of first linked object should be compatible with at least one interface of a second linked object, for link to be considered as valid. Allowed interaction types are described below.

All link types are "many-to-many", that is — a single object may participate in many links of same or of different types. However particular link types may add some logical constraints on involved objects' participation in other links. All of such cases are documented.

2.2.1 Video source – Video archive

Video sources, in particular RTSP video source and ONVIF device, can be linked with a video archive. In case if such link is established, the video archive performs permanent recording of data acquired from specified video source.

2.2.2 Video source – Recording controller

If a link is established between video source and a recording controller, the recording controller begins acquiring the video stream from that video source, and manages the recording of data from that video source into video archive. Video source whose data recording in a video archive is managed by recording controller should not be linked with the same video archive directly. Neither it should be linked with the same video archive via any other recording controller.

2.2.3 Recording controller – Video archive

If a link is established between recording controller and video archive, all video sources linked with that recording controller are implicitly linked with corresponding video archive. The difference with direct Video source – Video archive link is that the recording is controlled by recording controller (turned on and off upon requests to the controller) rather than takes place permanently while Viinex 3.0 is running.

2.2.4 Video source – Video renderer

A link established between a video source and the video renerer means that the renderer should be able to display the video from respective video source in its output. The actual content displayed by video renderer is controlled by its initial configuration and by HTTP calls to the video renderer object, as described in section 3.11.

2.2.5 Video source – Stream switch

A link established between a video source and the stream switch means that the switch should be able to translate the stream from respective video source as its own output. The actual stream translated by the stream switch as its output is defined by its configuration and by HTTP calls to the stream switch object described in section 3.12.

2.2.6 Video source – WebRTC server

If a link is established between a video source and WebRTC server, the former is published within the latter to be available for acquiring the live video stream. For details regarding getting access to a live video source by means of WebRTC, see section 3.14.

2.2.7 Video source – RTSP server

If a link is established between a video source and RTSP server, the former is published within the latter to be available for acquiring the live video stream via RTSP protocol. The RTSP URI for obtaining a video stream is defined by the TCP port which the RTSP server listens on, and the `name` parameter value of the video source object:

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

2.2.8 Video archive – RTSP server

If a link is established between a video archive and RTSP server, the former is published within the latter to be available for acquiring the stored video streams via RTSP protocol. The RTSP URI for obtaining a video stream 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`*`.`

2.2.9 Video source – Web server

If a link is established between a video source and a web server, the former is published under that server to be available for acquiring the live video data via HLS protocol and additional information on the video stream. The details for acquiring media data from a web server are described in section 3.4 of this document.

2.2.10 Event source – Web server

If a link is established between an event source and a web server, the events generated by that event source will be available to the WebSocket clients of that web server instance, i.e. such clients can subscribe to events from that objects and expect to receive that events. See section 3.19 for more details.

2.2.11 Snapshot source – Web server

If a link is established between a snapshot source and a web server, the former is published under that server to be available for acquiring still images (snapshots) via HTTP requests. The details for acquiring snapshots from a web server are described in section 3.8 of this document.

2.2.12 Overlay control – Web server

If a link is established between an overlay control and a web server, the former is published under that server to be available for controlling the content of the overlay drawn over the resulting video channel of the raw video source or the video renderer) by means of sending HTTP requests. The details for controlling the overlay content via HTTP API are described in section 3.9 of this document.

2.2.13 Layout control – Web server

If a link is established between a layout control and a web server, the former is published under that server to be available for controlling the layout of viewports on the respective video renderer instance via HTTP requests. The details for controlling the layout via HTTP API are described in section 3.11 of this document.

2.2.14 PTZ control – Web server

If a link is established between a PTZ control and a web server, the former is published under that server to be available for controlling the PTZ functionality on the respective ONVIF device via HTTP requests. The details for controlling the PTZ device via HTTP API are described in section 3.13 of this document.

Note that in order for the API of PTZ control to become availabe, it has to be explicitly enabled by means of the `enable` parameter in the configuration of ONVIF device in Viinex 3.0, as described in section 2.1.2.

2.2.15 WebRTC server – Web server

A link between a WebRTC server object and the webserver is necessary for the former to become reachable by remote clients for "signaling" purpose – that is, such link publishes the WebRTC server under the web server, so that clients can create new WebRTC sessions, get SDP offers and return SDP answers, – via the HTTP API. For more information refer to section 3.14.

2.2.16 Vehicle license plate recognizer

Vehicle license plate recognizer – Web server

If a link is established between LPR engine and Viinex 3.0 HTTP server, the former is published under the HTTP server to be available for remote calls (requests) for recognition. The details for such calls are described in API section of this document.

This connection is suitable for `alpr` objects functioning in modes `"image"` or `"any"`, and for `alprvideo` object. The API exposed by the two types of obects differs, see sections 3.15.1 and 3.15.2 respectively.

Vehicle license plate recognizer – Video source

If a link is established between LPR video and a video source, the former uses the latter to receive the video data for analysis.

It is required for the video LPR object to be linked against at least one video source. It is possible for video LPR object to be linked against two or more video sources. In such case, the video data from sources linked to the video LPR object is processed alternately (taking one frame from the first linked video source, then one frame from the second linked video source, and so on, — thus making a round-robin of the linked video sources).

LPR engine – Video LPR

A link established between an `alpr` object that is functioning in modes `"video"` or `"any"`, on one side, and an `alprvideo` object, on the other side, is necessary for the latter to function properly. The `alprvideo` object uses the `alpr` object as the recognition engine, and inherits all of its properties (like selected countries, transliteration and so on).

2.2.17 Face detection

Face detection engine – Web server

If a link is established between the face detection engine and Viinex 3.0 HTTP server, the former is published under the HTTP server to be available for remote calls (requests) for face detection. The details for such calls are described in API section of this document.

This connection is suitable for `facedet` objects functioning in modes `"image"` or `"any"`, and for `facedetvideo` object. The API exposed by the two types of obects differs, see sections 3.17.1 and 3.17.2 respectively.

Video face detector – Video source

If a link is established between the `facedetvideo` object and a video source, the former uses the latter to receive the video data for analysis.

It is required for the video face detector object to be linked against at least one video source. It is possible for video face detector object to be linked against two or more video sources. In such case, the video data from sources linked to the video LPR object is processed alternately (taking one frame from the first linked video source, then one frame from the second linked video source, and so on, — thus making a round-robin of the linked video sources).

Face detection engine – Video face detector

A link established between an `facedet` object that is functioning in modes `"video"` or `"any"`, on one side, and an `facedetvideo` object, on the other side, is necessary for the latter to function properly. The `facedetvideo` object uses the `facedet` object as the recognition engine, and inherits all of its properties (like selected countries, transliteration and so on).

2.2.18 Recording controller – Web server

If a link is established between recording controller and Viinex 3.0 HTTP server, the former is exposed under HTTP server to be available for remote requests to control (turn on and off) the recording of video sources linked with that controller, in the archive linked with that controller. For details on controlling video recording via HTTP requests see the API section of this document.

2.2.19 Recording controller – Rule

If a link is established between recording controller and a `rule` object, the latter takes control over when the recording should start and stop within the former. There can be several recording controllers linked to a single rule. However, as stated in 2.1.8, one recording controller should only be subordinated to a single source of commands, – that is, a web server (with a single client connecting it to issue commands for `recctl` object), or one rule. If a single recording controller is linked to more than one object (rule or web server), an error is reported on Viinex 3.0 startup.

2.2.20 Rule – Event source

The `rule` objects in Viinex 3.0 are driven by events coming from event sources, that is – from ONVIF devices. Each object of type `onvif` implements the event source interface, and can be connected with a `rule` object, in which case the latter receives and processes the events from the former. There is no limit on the number of links between one rule and different event sources, and vice versa, between a single event source and different rules. In other words, one rule is allowed to process events from multiple ONVIF devices, and one ONVIF device is allowed to participate in multiple rules.

2.2.21 Video archive – Web server

If a link is established between video archive and web server, the former is published under HTTP server to be available for remote calls for acquiring stored video data and video archive contents. The details for such calls are described in API section of this document.

2.2.22 Video archive – Replication source

If a link is established between video archive and a replication source, then all video channels stored in that video archive will be replicated by corresponding replication source.

This link is mandatory part of replication source configuration. If such link is not established for a replication source, Viinex 3.0 would fail at startup, writing corresponding error message to its log.

2.2.23 Video archive – Replication sink

If a link is established between video archive and a replication sink, then this video archive will be used by corresponding replication sink to store all video data coming from replication sources who will be connecting to that replication sink. Video channels will be created in the video archive dynamically in the runtime, as they become known to the replication sink from its peer replication sources.

This link is mandatory part of replication sink configuration. If such link is not established for a replication sink, Viinex 3.0 would fail at startup, writing corresponding error message to its log.

2.2.24 Replication sink – Web server

The link established between a replication sink and a web server defines which web server should be used to expose its private API of the replication sink to be used by peer replication sources. One replication sink may be exposed via one or more web servers. This provides flexibility for setting up a distributed configuration accessible in several subnets, each with its own set of credentials for replication sources' authentication.

2.3 Third-party video management systems

This section describes the configuration options for Viinex 3.0 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 2.1.13

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.

2.3.1 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 3.0 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 3.0 to authenticate at Milestone XProtect VMS instance.

> **NB!** In the context of interoperation with Milestone XProtect, Viinex 3.0 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 3.0 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 3.0 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 3.0 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 3.0 configuration, in the video channel selector property `channel.id` of respective `vmschan` object.

An example of Viinex 3.0 object for integration with Milestone XProtect is given below:

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

2.3.2 Geutebrück G-Core

An integration object for G-Core VMS by Geutebrück GmbH (https://www.geutebrueck.com/) is represented in Viinex 3.0 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 3.0.

G-Core integration in Viinex 3.0 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[14], 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 3.0 is the identification by means of `global_number` parameter.

An example of Viinex 3.0 object for integration with Geutebrück G-Core is given below:

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

2.3.3 Qognify (SeeTec) Cayuga

An integration object for Cayuga VMS by Qognify GmbH (former SeeTec), https://www.qognify.com/products/cayuga/, is represented in Viinex 3.0 by a configuration in the following format:

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

[14] 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.

NB! The integration of Cayuga VMS implementation in Viinex 3.0 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 3.0 is running.

Cayuga integration in Viinex 3.0 supports specifying a video stream using the name of the camera and the human-readable number of the video source. For using 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 using 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 3.0 object for integration with the Cayuga VMS is given below:

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

2.3.4 DSSL Trassir

An integration object for DSSL Trassir VMS (https://trassir.com/) is represented in Viinex 3.0 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 3.0 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 3.0 would connect to Trassir without checking the certificate. This scenario should be avoided in public networks.

Trassir integration in Viinex 3.0 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"
}
```

2.3.5 ITV|AxxonSoft Intellect

An integration object for ITV|AxxonSoft Intellect VMS (https://www.itv.ru/products/intellect/) is represented in Viinex 3.0 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 3.0 instance and Intellect instance are running in the same time zone.

Note that Intellect integration in Viinex 3.0 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"
}
```

2.4 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.

2.4.1 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.

2.4.2 Credentials database

This paragraph describes the section of configuration document that is common for HTTP and RTSP servers built into Viinex 3.0, and defines the database of credentials to authenticate clients accessing that servers.

An example for the credentials database section is given below:

```
"auth": {
  "require": true,
  "realm": "ViinexAuth",
  "htdigest": "C:/htdigest.txt",
  "accounts": [
    {
      "type": "password",
      "login": "admin",
      "digest": "510070c93040af4bef5e6d311d26af74"
    },
    ...
    {
      "type": "apikey",
      "key": "remoteagent1",
      "secret": "foobarsecret1"
    },
    ...
  ]
}
```

There are two mandatory parameters, `require` and `realm`, and two optional parameters, `htdigest` and `accounts`.

The `require` parameter defines whether the server should require clients' authentication at all. Setting it to `false` permits the anonymous access to the server instance.

When the parameter `require` is set to `true`, the server checks its clients' authentication against the credentials database stored under parameter `accounts`, or referred to by parameter `htdigest`, or both.

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",
  "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.

The `htdigest` parameter is another option to refer to credentials database. When specified, this parameter 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 3.0 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 3.0 startup. Viinex 3.0 also does not recognize the changes to htdigest file while running: in order for such changes to take effect, the Viinex 3.0 instance should be restarted.

If both `accounts` and `htdigest` options are provided, the resulting credentials database would be the union of set of records provided in `accounts` array and the content of referred `htdigest` file.

In both cases the realm is required to compute digest. Realm is an arbitrary string value that should be provided in the parameter `realm`. Realm is usually used to distinguish records in credentials databases: records with different realms are not interchangeable, and the credentials computed with some realm is useless if it mismatches the realm used by the server. Note that there can be only one realm for one instance of the Viinex 3.0 HTTP or RTSP server (the one that is specified by `realm` parameter), while there can be records with many different realms stored in one htdigest file. Viinex 3.0 ignores the records in htdigest file with realm string mismatching the value of `realm` parameter specified in the server configuration.

2.4.3 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 3.0 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 3.0 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 3.0 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 3.0 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 3.0 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 3.0. 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 3.0 may log a warning but proceeds.

2.4.4 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",
    "quality": STRING,
    "profile": STRING,
    "preset": STRING
}
```

The `type` defines the type of encoder implementation. Allowed value for that parameter is `"cpu"`, which stands for software encoder implementation.

`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).

2.4.5 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 3.0 uses the `wkhtmltoimage` to render HTML. `wkhtmltoimage` is distributed with Viinex 3.0 as a separate binary (Viinex 3.0 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 3.0 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 3.0 startup. This property may hold a path to HTML or BMP file. Viinex 3.0 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 3.0 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 3.0 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 3.0 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.

2.4.6 Analytics engine

The computer vision engines built into Viinex, namely the license plate recognition engine and the face detection engine, share several common settings. The configuration sections for mentioned modules should have form of:

```
{
    "type": STRING,
    "name": STRING,
    "mode": "image" | "video" | "any",
    "datapath": STRING,
    "workers": INTEGER
    ...
}
```

where the ellipsis "`...`" denotes the application-specific parameters described for the license plate recognition engine and for the face detection engine in their respective sections 2.1.14, 2.1.15.

Both of mentioned modules behave identically in the sense that they provide the functionality for the analysis of a still images, allowing this feature to be used "as is", as well as in a

combination with respective video analytics module to produce results on video sequences. How exactly the analytics feature is available depends on the `mode` parameter.

The `mode` parameter of the configuration object is an optional string taking one of three possible values: ``image'', ``video'', or ``any''. These values define how the instance of the analytics engine can be used: whether it can only be published in a web server to be accessible via HTTP API to perform still image processing upon external request (the ``image'' case), or it can only be attached to one or more instances of objects performing the processing on video sequences (see section 2.4.7), which is the case of ``video'', or it can be used in both scenarios (``any''). By default, if no value for `mode` property is specified, the value ``image'' is assumed. The value for `mode` configuration parameter cannot be defined arbitrarily by user; it should be agreed with what is allowed in the license key – it is is checked against the license key, and an error is issued at startup if the ``mode'' which is not permitted by license is specified.

The `datapath` parameter should contain the path to immutable data files that are used by the analytics engine (LPR and face detection engines have each their own set of such immutable data). This field is typically populated with correct values during standard deployment (Windows Installer or Debian package manager) and needs not to be modified by the user.

The `workers` parameter directly affects the throughput of the analytics engine instance. It specifies the number of requests for image processing which can be processed simultaneously (in parallel). Of course this also depends on the actual parallelism of the hardware platform, but the `workers` parameter defines how many CPU cores, at maximum, can be utilized by the instance of the engine. The `workers` parameter cannot be set arbitrarily and need to be coherent with what is allowed in license key (that is, the workers number set in configuration is checked against license key).

2.4.7 Video analytics module

The video analytics modules built into Viinex, namely the license plate recognition module (see section 2.1.14) and the face detection module (see section 2.1.15), both share a common set of configuration parameters. These modules implement the common behavior: they are linked to one or more video sources, they expect an external signal (HTTP request) to perform the video analysis and output the result as the response to that request. Both of that modules need to be linked with an "engine" module, which performs the actual analysis. Such engine is the license plate recognition engine (`"type": "alpr"`) and the face detection engine (`"type": "facedet"`) respectively.

The configuration section for video analytics module should have the following form:

```
{
    "type": STRING,
    "name": STRING,
    "freeflow": BOOLEAN,
    "preprocess": FLOAT,
    "postprocess": FLOAT,
    "skip": INTEGER | "non_idr" | "while_busy",
    "time_budget": FLOAT,
    "roi": [FLOAT, FLOAT, FLOAT, FLOAT],
    "confidence": FLOAT,
    "snapshots": INTEGER,
    "snapshots_dump_path": STRING
}
```

The description of that configuration parameters is given below. All of the parameters are optional, except the `type` and `name` parameters whch Viinex requires to have in every modules' instance configuration section.

The `freeflow` parameter can be set to `true` in order to override the operation mode of the video analytics module in the way so that, instead of waiting of a command to perform analytics from the API, and responding with the analysis result to that command, – with `"freeflow":true` the module is performing the analysis continuously, without external commands. There are two consequences if the `freeflow` mode is enabled: 1) the commands to initiate the video analytics are no longer accepted by such module. It does not expose respective methods in its HTTP API. 2) The instance of module becomes an event source; it sends the results of a video analysis as events. Therefore such video analytics module in `freeflow` mode should be linked with event consumers (like web server or a script) in order to process or further pass the resulting events. The default operation mode, when this parameter is not set, is to wait for a request to perform video analytics, and return the result as a response to that request, – which corresponds to the default value of `freeflow` parameter equal to `false`.

The `preprocess` and `postprocess` values are numbers defining how deep into the past and how far into the future the video should be analyzed with respect to the moment when a command for video analysis is received via HTTP API. If the request for video analysis is received at moment t_0, assuming values `preprocess` and `postprocess` to be equal to $\tau_{\texttt{preprocess}}$ and $\tau_{\texttt{postprocess}}$ respectively, the video data from interval

$$t : t_0 - \tau_{\texttt{preprocess}} \leq t \leq t_0 + \tau_{\texttt{postprocess}}$$

is going to be analyzed. The `preprocess` and `postprocess` are expected in seconds. In the example given above, `preprocess` equals to 1.5 seconds and `postprocess` equals to 1 second, so totally at least 2.5 seconds of video should be available for recognition when a request is received[15].

When not set, the `preprocess` and `postprocess` parameters are assumed to have the default value of 0, which means that only one frame should be processed upon each request (the one that is most close to the moment when the request was received).

The `skip` parameter defines how H264 video stream is handled to present the frames to the video analytics engine which performs computer vision analysis on still images. This parameter may take a an integer value which describes how the video stream is decimated. `"skip": 0` means the stream is not decimated (0 frames are skipped, every frame is processed), and every frame within $[t_0 - \tau_{\texttt{preprocess}}, t_0 + \tau_{\texttt{postprocess}}]$ interval is presented to license plate recognizer. `"skip":1` means that after presenting one frame to recognizer, 1 frame is skipped. Likewise, `"skip": N` means that after presenting one frame to recognizer, next N frames are skipped, that is — every $N+1$-th frame is presented to the analytics engine. The `skip` parameter may also take two special string values: `"non_idr"`, which means that all frames except IDR (i.e. keyframes) are omitted, and `"while_busy"`, which stands for skipping everything while the previous video frame is being processed (and then, of course, one has to wait for the next IDR frame to decode and process it).

The `skip` parameter helps saving CPU resources, not waisting them on analyzing the frames that were shot within a close amount of time, and are likely similar and therefore not containing new information in comparison with each other. Note however, that to implement

[15]The actual length of video buffered may be greater than given values, depending on the GOP size (the interval between keyframes). Since Viinex 3.0 handles H264 video, it is necessary to have the full GOP, from the previous IDR frame, to decode an image with specific timestamp. Therefore, if GOP size equals to 1 second, the amount of video that is actually buffered may be equal to GOP size + preprocess + postprocess = 1.0 + 1.5 + 1.0 = 3.5 seconds.

the decimation strategy, Viinex 3.0 needs to decode all the video received from video souce (because failing to decode a single frame within a GOP results in an impossibility to decode the rest of that GOP). That means that decimation only saves resources on video recognition, not on decoding. The value `"skip": "non_idr"` provides the most economical strategy for data processing which allows to not lose arbitrarily long video fragments even while the CV engine is busy: the frames are enqueued to be processed. It is recommended choice, if suitable in other aspects of usage scenario (i.e. if vehicles move slow enough for the license plate text to be readable on at least one IDR frame within $[t_0 - \tau_{\texttt{preprocess}}, t_0 + \tau_{\texttt{postprocess}}]$ interval). The value `"skip": "while_busy"`, on the other hand, is similar to the `"non_idr"` and is even more economical because it guarantees that there is no data queued to be processed. It therefore guarantees that the processing ends as soon as the $t_0 + \tau_{\texttt{postprocess}}$ timestamp is reached in the input video stream. Last but not least, the `"skip": "while_busy"` setting might be the only viable option when the video analytics module is linked with a raw, uncompressed video source (namely the USB camera), the time required for processing of every frame is significantly larger than the iter-frame period, and the `postprocess` time is long. In described situation, if the video data is queued to be processed with a guarantee, it is possible that the queue grows rapidly, and with uncompressed video this is highly undesireable[16]. Therefore it is recommended that `"skip": "while_busy"` is used whenever video analytics module is linked with a raw video source (otherwise, an additional care should be taken to make sure that video registration does not suffer from data queueing in video analytics module).

When not set, the `skip` parameter is assumed to have the default value of `"non_idr"`.

The `time_budget` setting is certain way connected with `preprocess`, `postprocess` and `skip`, but relates to control over the process of the video analysis rather than to the data being analysed. Indeed, it is worth to be emphasized that while the `preprocess` and `postprocess` parameters denote the time interval, that interval is measured against the timeline within the video stream being processed. It has nothing to do with the real time clock on the system where the analysis is performed. In contrast, the `time_budget` denotes the time allotted for video processing. That is a real time interval, which is measured against the "wall clock". Whenever the analytics module realizes that the time spent for video analysis on a single request exceeds the value of the `time_budget` parameter (in seconds), – it stops taking new input frames for processing, and returns the result as soon as the frames that already appeared on the "processing converyor" are analysed. Because of the presence of the "processing converyor" internally in the video analytics module, the `time_budget` parameter should not be considered as a very precise guaranteed time to answer (the converyor length might be added to that time on practice), but it provides a "fuse" for the case if CPU resources are scarce and/or the processing does not terminate early because of reaching sufficient result confidence (see below). Note that it typically does not make sense to set the `postprocess` value less than `time_budget`.

When the `time_budget` parameter is not set, no time limit is enforced for processing the data within configured time interval upon each request.

The `roi` parameter is optional and defines the region of interest on the frame where the video analysis should be performed. This parameter should have the form of JSON array of four

[16]The video device drivers often cannot deal with arbitrary number of memory buffers to grab the video into. This is connected with receiving the data from the video device via DMA. There is typically a small number of raw video buffers (frames) which userspace code can hold without video grabber having to suspend the video registration; and then the userspace code should "return" the buffers back to the driver, in order for video registration to be resumed. As a result, it is generally not possible to queue many raw video frames in the userspace for time periods more than several hundreds of seconds, without the need to copy the uncompressed video data as soon as possible. Such copying itself introduces noticeable overhead, for which reason it is not performed in Viinex.

floating-point numbers in range $[0, 1]$, denoting left, top, right and bottom coordinates of ROI rectangle on the original image, with respect to its width (for left and right) and height (for top and bottom). This parameter may be used to additionally speed up processing, and to get rid of subtitles on an image, which can interfere with vehicle license plate recognition process. When `roi` is not set, is is assumed that the whole image should be processed.

The `confidence` parameter defines the sufficient level of confidence which signals that processing of further video frames is not required for that processing request. After the request for processing is received via HTTP API, the video analytics module begins presenting the video frames to its respective analytics engine sequentially, frame by frame, in order of increasing timestamp. This process breaks after the first result with `confidence` greater or equal to specified value is produced by the analytics engine. By default, value of `"confidence": 0` is assumed, i.e. processing stops when some (any) result is formed.

The `snapshots` value defines how many recent snapshots should be available to retreive via HTTP API from this instance of video analytics module (see section 3.15.3, 3.17.3). When the processing result is returned via HTTP API, it contains the timestamp of a video frame where the result was obtained. This timestamp may be used in another HTTP API request to obtain that video frame in form of a snapshot (JPEG image). Setting the `snapshots` parameter guaranees that corresponding number of most recent snapshots are available. That snapshots are held in memory, so it's not recommended to set this value to a large number: make it sufficient for you usage scenario. Older snapshots are evicted from memory (overwritten). The default value for this parameter is 10.

There is also the `snapshots_dump_path` parameter in the presence of which the video analytics module is instructed to write out the "best frames" to the local filesystem. The only use case this option is intended for is to gather the information for image analytics algorithms' quality estimation. This option should never be used in production environment: it involves additional overhead of an extra step of unneeded image compression, and there is also no means for the disk space used by stored images to be automatically limited.

2.4.8 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.

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 3.0 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 3.0 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.

2.5 License information

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. When the license `license` key is absent, license information is taken from USB dongles or other sources.

The value stored under `license` key of the configuration document should be a "license document" – a string containing the encrypted license information together with information about hardware where an instance of Viinex 3.0 is allowed to run on. License documents are generated by Viinex 3.0 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 6.2. 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 3.0 is supposed to run, and they cannot be used to update a USB dongle.

2.6 Split configuration

For convenience of deployment and for better manageability, the configuration document for Viinex 3.0 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 3.0 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 3.0 is available for use from client applications by means of HTTP API. When a component of Viinex 3.0 is published in the built-in Viinex 3.0 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 3.0 components.

All components' API are published under URLs depending on the names of components in Viinex 3.0 configuration document. For instance, when a vehicle license plate recognizer with name "alpr0" is published in the web server, its API becomes available under subtree

`http://SERVERNAME:SERVERPORT/v1/svc/alpr0`

This is a general rule: the common prefix of `/v1/svc/`*`NAME`* is prepended to the URLs for accessing functionality of Viinex 3.0 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 3.0 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 interface types.

Request URL and applicable methods

`GET http://servername`*`:port`*`/v1/svc`

Request parameters

none

Response syntax

JSON array of 2-element arrays (tuples):

```
[ ["interfaceType1", "componentName1"],
  ...,
  ["interfaceTypeN", "componentNameN"] ]
```

Response example

```
[
  ["AutoLPR", "alpr0"],
  ["VideoStorage", "stor0"],
  ["VideoSource", "cam1"],
  ["VideoSource", "cam2"],
  ["SnapshotSource", "cam2"]
]
```

means that there are four components published under the web server: a vehicle license plate recognizer, a video archive, and two video sources. Note that while Viinex 3.0 configuration file contains definition for object types, these are essentially implementation types. One object may, however, implement several interfaces, which can be exposed (or not exposed) in HTTP API. In this request, interface types are described, not implementation types.

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 3.0 object `name`s, and values equal to the content of `meta` property in configuration of respective Viinex 3.0 object:

```
{
  "componentName1": JSON_VALUE_1,
  ...
  "componentNameN": JSON_VALUE_N
}
```

Only the components are reported which are published under this instance of Viinex 3.0 web server, and which have their `meta` property set to a non-null JSON value.

Response example

For the configuration example given at page 9, 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 3.0 configuration to decide how to use these values. Viinex 3.0 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

`GET 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,
    "salt": 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 `salt` string is optional and returned only in case if the `login` parameter referenced the account with `password` authentication data record type, see section 2.1.21 for details. `salt` 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, `salt` 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. Authentication response is JSON document with fields `login, when, challenge, signature` and `response`. The first four fields have the same meaning as in challenge description. The `response` field is HMAC [7] of `challenge` data, computed with client's `secret`, if the authentication data record type is `apikey`, or salted and hashed password, if authentication data record type is `password` (see section 2.1.21 for details.).

Response syntax

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 is base64-encoded JSON document containing that information. Simultaneously, the same document is returned in `authCheckResponse` response body.

```
{
    "salt": STRING,
    "issued": TIMESTAMP,
    "user": INTEGER,
    "sign": STRING
}
```

Upon authentication failure, server returns HTTP error code 401.

Response example

Request:

```
{
    "login": "agentA",
    "challenge": "fdfeefeaa33d4f683bc843cae4375592439fa980ac969e\
                  7757226baf15ef5398",
    "when":"2017-01-16T12:06:46Z",
    "signature": "ffad8f01f7ad42742a9e263c34e46b4d",
    "response": "4c7bd5ae85894f78eb87bd2955f4cd83"
}
```

Response:

```
HTTP/1.1 200 OK
Set-Cookie: auth=eyJzYWx0IjoiMzQxNDcwMzE5ZjVlZDRhODIxM2UzMjdhMWU\
                  yNTJiODJlYzhhZmFmZGM4MWYzMzQxNTNjZmE5YzU5YmFlMm\
                  NhMSIsImlzc3VlZCI6IjIwMTctMDEtMTZUMTI6NTg6MzkuN\
                  zcyODI0NFoiLCJzaWduIjoiYWMzMmZkYTc1NDViZDlhNGEx\
                  MTQ5YzRjOWVjOTkzNWEiLCJ1c2VyIjoxfQ==


{
```

```
    "salt": "05df034adb47f865bfba4bb5028660d8fa3fd6774f8a391c83a\
             2437737062afa",
    "issued": "2017-01-16T12:57:34.8826421Z",
    "user": 1,
    "sign": "3c60158187b07dfbb972d084e7c7831e"
}
```

The client should compute its `response` to challenge as HMAC-MD5 [7] of the `challenge` data. The secret for computing the HMAC is `secret` or MD5 hash of salted user's password (in such case, the salt 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 3.0 public API. To turn authentication off, one may set parameter `auth.require` to value `false` in configuration of corresponding `webserver`, as described in section 2.1.21.

3.3 Environment

This subsection contains description for API calls for finding out the details of environment where Viinex 3.0 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 3.0. `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 6.2.1, 6.2.2). 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 3.0 video channel in a 3rd party VMS;
- `LPRecognizer` – an instance of license plate recognition engine;
- `ReplicationSink` – video archive replication sink (i.e. the server capable of gathering video information from multiple Viinex 3.0 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,
      "LPRecognizer": 2,
      "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, 2 video replication sources, and 2 vehicle license plate recognizers.

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 a JSON array of exactly two elements, – a string and an integer:

```
[ [STRING1, INT1], [STRING2, INT2], ... ].
```

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 3.0 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 3.0 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 3.0 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 3.0 software version

Request purpose

Obtain the information on Viinex 3.0 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 3.0, 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 3.0.

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 3.0 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 3.0 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 2.4.3.

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 3.0; `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
```

3.5 Video archive

Assuming the video archive component has the name `storageN` in Viinex 3.0 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_boundaties": [TIMESTAMP, TIMESTAMP],
      "disk_usage": INTEGER
    },
    ...,
    "videosourceN": {
      "time_boundaties": [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 3.0 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/videosourceM/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).

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/videosourceM/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
```

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 2.1.7.

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 3.0 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:

```
{
  "type": "rtsp" | "vmschan" | "mediafile",
  "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,
  "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 3.0 as described in section 2.1.13. 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 3.0 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 3.0 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 3.0 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
{
    "type": "rtsp",
    "meta": {},
    "url": "rtsp://192.168.0.71:554/stor0/cam2?
begin=2017-06-21T08:36:19.365Z&end=2017-06-21T08:37:11.327Z",
    "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 3.0). 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`[1] 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

[1] Which was misspelled as `canceled` in Viinex builds prior to 2.0.0.326 but fixed after that. The backwards compatibility with misspelled status name ''`canceled`'' was not preserved.

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 3.0 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 3.0 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[2], 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 3.0 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 3.0 returns an error. The `cached=N` parameter, when used in request to a video archive, instructs Viinex 3.0 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 source will be used in this request. When no parameters are given, Viinex 3.0 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

[2] 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.

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 800×600, then ROI left-top corner is $(160, 180)$, and ROI size is 280×90 (measured in pixels on the original image).

The `scale=N` parameter instructs Viinex 3.0 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 3.0 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 3.0 preserves the aspect ratio of the image, no matter what spatial-related parameters are specified. If only `width` or only `height` is set, – Viinex 3.0 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 3.0 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 800×600, and `scale=4` specified, the resulting image will have the size of 200×150. 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 3.0 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 320×240 instead of requested 320×200.

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 2.4.5.

3.10 Video renderer

When published under the Viinex 3.0 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 2.4.8. 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 1280×960, 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[3].

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 . . . 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.

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]'
```

[3]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.

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 3.18.2.

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 3.0 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 3.0 currently does not support certain features, most noticeable of which are preset tours.

Viinex 3.0 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 3.0 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.[4] 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].

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,
```

[4] 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.

```
    "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 3.0 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 3.0 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 3.0 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 2.1.20, 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 3.0 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 3.0 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 3.0 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 3.0 object with respective name as a video source, or a JSON object with property `name` and its value holding the name of Viinex 3.0 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 2.1.13 for more details), Viinex 3.0 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 3.0 server.

There is also another syntax for this request, which was introduced initially, when WebRTC-related functionality was at first implemented in Viinex 3.0. 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 3.0 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 3.0 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 3.0 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 3.0 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 3.0 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
}
```

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 3.0 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.

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 3.0 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 Vehicle license plate recognition

3.15.1 Perform recognition on a given still image

Request purpose

Find and recognize vehicle license plates on an image using recognizer with name `alprN`; return the results.

Request URL and applicable methods

`POST http://servername:port/v1/svc/`*`alprN`*

Request parameters

Image to be processed must be passed as POST request body without additional containers (encoding layers). To be processed, an image should be in JPEG format.

Response syntax

JSON array of objects – recognition results. May be empty, if no license plates were found.

```
[{"plate_text":STRING,
  "confidence":FLOAT,
  "plate_rect":{"left":FLOAT,"top":FLOAT,
                "right":FLOAT,"bottom":FLOAT}},
 ...
]
```

where

`plate_text` – text read from license plate by recognizer;

`confidence` – an estimation of this results' likelihood, percents;

`plate_rect` – an object containing floating-point numbers for left-top corner and bottom-right corner of rectangle fitting the license plate on the image. The coordinates are given in relative units (from 0 to 1); to obtain coordinates in pixels, one should multiply the relative values by image width or height respectively.

Response example

A correct request can be sent using CURL utility (https://curl.haxx.se/download.html):

```
curl -X POST http://localhost:8880/v1/svc/alpr0
    --data-binary @FILENAME.jpg
```

Response:

```
[{"plate_text":"BC6841BE",
  "confidence":86.78,
  "plate_rect":{"left":0.639,"top":0.852,
                "right":0.708,"bottom":0.878}}
]
```

— one license plate was found with text "BC6841BE"; hypothesis confidence is about 87%, relative license plate coordinates on an image are returned.

3.15.2 Perform recognition on a video source

Request purpose

Find and recognize vehicle license plates on video using `alprvideo` object with name `alprCamN`; return the results.

Request URL and applicable methods

`GET http://servername:port/v1/svc/`*`alprCamN`*

Request parameters

none

Response syntax

JSON array of objects – recognition results, or, if no license plates were found, an array of one element - a JSON object containing single property `timestamp`.

```
[{"plate_text":STRING,
  "confidence":FLOAT,
  "plate_rect":{"left":FLOAT,"top":FLOAT,
                "right":FLOAT,"bottom":FLOAT}},
  "timestamp": TIMESTAMP
 },
 ...
]
```

where `plate_text`, `confidence` and `plate_rect` fields have same meaning as in sec. 3.15.1, and `timestamp` field contains the timestamp of the video frame where the best or sufficient hypothesis was obtained. This value can be used in a request to a snapshot (see sec. 3.15.3). If no recognition results were produced, an answer would look like this:

```
[{ "timestamp": TIMESTAMP }]
```

where the timestamp of the frame stored as the snapshot is returned.

Response example

A correct request can be sent using CURL utility (https://curl.haxx.se/download.html):

```
curl -X GET http://localhost:8880/v1/svc/alprCam1
```

Response:

```
[{"plate_text":"BC6841BE",
  "confidence":86.78,
  "plate_rect":{"left":0.639,"top":0.852,
                "right":0.708,"bottom":0.878},
  "timestamp": "2017-10-08T21:00:53.231Z"}
]
```

— one license plate was found with text "BC6841BE"; hypothesis confidence is about 87%, relative license plate coordinates on an image are returned. The timestamp of a frame where the best (or sufficient) hypothesis was produced is retutned in the `timestamp` field, which can be used in a request for corresponding snapshot.

3.15.3 Obtain a snapshot of a recently recognized vehicle

Request purpose

Obtain a snapshot of the vehicle with recognized license plate using `alprvideo` object with name `alprCamN`.

Request URL and applicable methods

`GET http://servername:port/v1/svc/`*`alprCamN/snapshot`*

Request parameters

`timestamp` – one mandatory parameter, referring to exact timestamp of the frame where the license plate text was best recognized. The timestamp may be given in ISO 8601 format, or as an integer number of milliseconds elapsed since UNIX epoch.

Other parameters are optional and correspond to description given in section 3.8.

Response example

A correct request can be sent using CURL utility (https://curl.haxx.se/download.html):

```
curl -X "GET http://localhost:8880/v1/svc/alprCam1/snapshot?\
timestamp=2017-10-08T21:00:53.231Z&\
roi=(0.639,0.852,0.708,0.878)"
```

Response – a JPEG image of the license plate recognized in the example of request given in section 3.15.2.

Remarks

Note that snapshots of recognized vehicles are stored in RAM, and their number is limited by the `snapshots` parameter in the configuration of the `alprvideo` object. Older snapshots are overwritten by more recent ones as they appear.

Snapshots are stored whenever a request for license plate recognition is issued, no matter whether some recognition results were produced or not. If no results were produced, the `alprvideo` object stores a frame coming right after the time when the request was received by Viinex 3.0 as the snapshot. The timestamp of that frame is reported in the `timestamp` property of respective API response (see section 3.15.2).

3.16 Railcar identification number recognition

The object `ridrcons` which is build into Viinex 3.0 for controlling the video analytics modules for railcar identification number recognition and consolidates the result from such analytics,

exposes the interface `Updateable` described in 3.18.2. The update call for `ridrcons` object serves for the purpose of starting and stopping of ID number recognition for a single railcar.

A JSON value passed to the update API call for `ridrcons` object should have the form of

```
{
    "recognize": BOOLEAN,
    "cookie": NUMBER
}
```

The `recognize` property is mandatory and defines whether the recognition of an ID number for the next railcar should start or stop. The `cookie` value is optional and may be used in order to distinguish the recognition results.

The video frames obtained by `ridrcons` object and its underlying video analytics processes after the update call with `recognize` property set to `true` are processed and the results of such processing are accumulated. When the `ridrcons` gets the update call with `recognize` property set to `false`, a consolidated recognition result is formed and sent to the event channel exposed by the `ridrcons` object. For simplicity, the same recognition result is returned as the result of respective update call. This may greatly simplify the integration in certain scenarios, because the recognition result does not need to be caught as an event, but rather can be obtained by software initiated the recognition in a synchronous manner.

The format of consolidated recognition result has the form of

```
{
    "cookie": INTEGER,
    "result": STRING,
    "confidence": FLOAT,
    "channels":[
        {
            "video_source": STRING,
            "result": STRING,
            "confidence": FLOAT,
            "timestamps":{
                "first": TIMESTAMP,
                "last": TIMESTAMP,
                "best": TIMESTAMP
            },
            "rect":[FLOAT,FLOAT,FLOAT,FLOAT]
        },
        ...
    ]
}
```

where the property `cookie` contains the cookie value passed to the `ridrcons` when recognition was started; `result` contains the recognition result for railcar identification number; `confidence` is a floating-point number specifying how confident is the algorighm on recognition results (this is a value in some ordinal scale; the bigger is the value, the higher is the confidence). The array `channels` contains the information on recogntion results for each video channel: the name of video source in `video_source` property, the recognition result for this specific video channel in a `result` property, the recogntion confidence for this specific video source, and the timestamps of the first frame, last frame of that railcar, and the timestamp

of the frame when the recognition result was seen by algorithm as "best". For this best frame, the rectangle which contains the recognized railcar number, is specified by the property `rect`. Note that the `best` and `rect` properties may be both `null` if no result was obtained on a video source for specific railcar.

An example for railcar number recognition result from the `ridrcons` object is given below:

```
{
    "cookie":52,
    "result":"60556339",
    "confidence":0.98,
    "channels":[
        {
            "video_source":"cam1",
            "result":"60556339",
            "confidence":0.98,
            "timestamps":{
                "first":"2020-07-21T15:14:52.822Z",
                "last":"2020-07-21T15:14:56.942Z",
                "best":"2020-07-21T15:14:53.982Z"
            },
            "rect":[0.2054,0.4208,0.7054,0.6291]
        },
        {
            "video_source":"cam2",
            "result":"60556339",
            "confidence":0.98,
            "timestamps":{
                "first":"2020-07-21T15:14:52.823Z",
                "last":"2020-07-21T15:14:56.943Z",
                "best":"2020-07-21T15:14:53.983Z"
            },
            "rect":[0.2054,0.4208,0.7054,0.6291]
        }
    ],
}
```

3.17 Face detection

3.17.1 Perform face detection on a given still image

Request purpose

Find and localize human faces on an image using the face detector instance with name `facedetN`; return the results.

Request URL and applicable methods

`POST http://servername:port/v1/svc/`*`facedetN`*

Request parameters

Image to be processed must be passed as POST request body without additional containers (encoding layers). To be processed, an image should be in JPEG format.

Response syntax

JSON array of objects – face detecton results. May be empty, if no faces were found on the given image.

```
[{"confidence":FLOAT,
  "face_rect":{"left":FLOAT,"top":FLOAT,
               "right":FLOAT,"bottom":FLOAT}},
 ...
]
```

where

`confidence` – an estimation of this results' likelihood, a number within a range of $[0, 1]$;

`face_rect` – an object containing floating-point numbers for left-top corner and bottom-right corner of a bounding box for each detected face. The coordinates are given in relative units (from 0 to 1); to obtain coordinates in pixels, one should multiply the relative values by image width or height respectively.

Response example

A correct request can be sent using CURL utility (https://curl.haxx.se/download.html):

```
curl -X POST http://localhost:8880/v1/svc/facedet0
    --data-binary @FILENAME.jpg
```

Response:

```
[{"confidence":0.9997726,
  "face_rect":{"left":0.33510125,"top":0.11539618,
               "right":0.5989618,"bottom":0.7536577}}]
]
```

— one face was found on the input image with the likelyhood about 99.97%; relative face bounding box' coordinates on an image are returned.

3.17.2 Perform face detection on a video sequence

Request purpose

Find and localize human faces on a video sequence using the `facedetvideo` object with name `faceDetCamN`; return the results.

Request URL and applicable methods

`GET http://servername:port/v1/svc/`*`faceDetCamN`*

Request parameters

none

Response syntax

JSON array of objects – face detection results, or, if no license plates were found, an array of one element - a JSON object containing the single property `timestamp`.

```
[{"confidence":FLOAT,
  "face_rect":{"left":FLOAT,"top":FLOAT,
               "right":FLOAT,"bottom":FLOAT}},
  "timestamp": TIMESTAMP
 },
 ...
]
```

The `confidence` and `face_rect` fields have same meaning as in sec. 3.17.1, and the `timestamp` field contains the timestamp of the video frame where the best or sufficient confidence was obtained for the detection results. This value can be used in a request to a snapshot (see sec. 3.17.3). If no recognition results were produced, an answer would look like this:

```
[{ "timestamp": TIMESTAMP }]
```

where the timestamp of the frame stored as the snapshot is returned.

Response example

A correct request can be sent using CURL utility (https://curl.haxx.se/download.html):

```
curl -X GET http://localhost:8880/v1/svc/faceDetCam1
```

Response:

```
[{"confidence":0.9997726,
  "face_rect":{"left":0.33510125,"top":0.11539618,
               "right":0.5989618,"bottom":0.7536577},
  "timestamp": "2017-10-08T21:00:53.231Z"}
]
```

— one face was found on the input image with the likelyhood about 99.97%; relative face bounding box' coordinates on an image are returned. The timestamp of a frame where the best (or sufficient) confidence was produced with face detection results is retutned in the `timestamp` field, which can be used in a request for the corresponding snapshot.

3.17.3 Obtain a snapshot of a recently detected face

Request purpose

Obtain a snapshot of the face using `facedetvideo` object with name `faceDetCamN`.

Request URL and applicable methods

`GET http://servername:port/v1/svc/`*`faceDetCamN/snapshot`*

Request parameters

`timestamp` – one mandatory parameter, referring to exact timestamp of the frame where the face was detected with the best confidence. The timestamp may be given in ISO 8601 format, or as an integer number of milliseconds elapsed since UNIX epoch.

Other parameters are optional and correspond to description given in section 3.8.

Response example

A correct request can be sent using CURL utility (https://curl.haxx.se/download.html):

```
curl -X "GET http://localhost:8880/v1/svc/faceDetCam1/snapshot?\
timestamp=2017-10-08T21:00:53.231Z&\
roi=(0.335,0.115,0.599,0.754)"
```

Response – a JPEG image of the license plate recognized in the example of request given in section 3.17.2.

Remarks

Note that snapshots of recognized vehicles are stored in RAM, and their number is limited by the `snapshots` parameter in the configuration of the `facedetvideo` object. Older snapshots are overwritten by more recent ones as they appear.

Snapshots are stored whenever a request for face detection is issued, no matter whether some detection results were produced or not. If no results were produced, the `facedetvideo` object stores a frame coming right after the time when the request was received by Viinex 3.0 as the snapshot. The timestamp of that frame is reported in the `timestamp` property of respective API response (see section 3.17.2).

3.18 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 3.0 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.

3.18.1 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 3.0. 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 3.0 as immutable data, till the next JavaScript call to `vnx.publish()`.

3.18.2 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 3.18.1).

3.19 WebSocket interface

In order to acquire real-time events from Viinex 3.0, the latter implements additional application-level interface on top of WebSocket protocol.

In order to use the WebSocket interface implemented by Viinex 3.0, 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 3.0 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 3.0 is straightforward. The client application can issue commands to alter its state, and can receive events from Viinex 3.0 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 3.0 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 3.0 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 3.0 client sends the heartbeat message to the server every 20 seconds. If Viinex 3.0 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 3.0 web server is configured to require authentication. With this request, the client should pass the value of `auth` cookie set by Viinex 3.0 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 3.0 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 3.0 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 3.0 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 3.0 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 2.1.9. 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 3.0 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 3.0 web server instance which is linked with that camera, as described in section 2.2 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 3.0 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 2.1.9 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

3.20 Configuration clusters

Configuration clusters are the way to create and dispose groups of Viinex 3.0 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 3.0 instance. That is, each cluster resemble an instance of Viinex 3.0 with some custom static configuration, with the difference that it can be created and destroyed in the runtime, without restarting the instance of Viinex 3.0. 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 3.0 instance and thus should be re-created explicitly every time such restart is done.

3.20.1 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"]
```

3.20.2 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 2.1.22.

3.20.3 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.

3.20.4 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.

3.20.5 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.

3.20.6 Access Viinex 3.0 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 3.0, such objects should be linked with an object of type `publish` which should be declared in their cluster. For more information refer to section 2.1.22.

3.20.7 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 3.0 WebSocket server is performed as described in section 3.19. 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.

4 Scripting and JS API

Viinex 3.0 implements scripting by means of introducing an object of type `script`, every instance of which represens an independent JavaScript execution context. Section 2.1.17 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 3.0 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 3.0 is documented. Section 4.3 goes into the detail of JS API provided by Viinex 3.0 objects that can be controlled from scripts.

4.1 Execution model and handlers

Like it was mentioned in section 2.1.17, scripts serve for the following purposes in Viinex 3.0:

- maintain internal state according to a custom logic, and expose a part of that state via HTTP API (see section 3.18.1);
- accept requests to update the internal state and reply to such requests to the parties who initiate them (see section 3.18.2);
- 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 3.0 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 3.0 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 semantics is explained in the following subsections. It is important that none of these handlers should not 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 for itself a timer.

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 3.0-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 3.0 configuration.

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 **??** 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 3.0 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 3.19.

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 3.18.2.

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 3.0 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 3.0 automatically serves HTTP requests to get this state, and that does not require attention from JavaScript code.

4.2 General puropose functions

In order to communicate with Viinex 3.0 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 3.0-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 3.18.1.

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.

4.2.3 `vnx.event()`

If a script is linked in Viinex 3.0 configuration with other objects that implement the event consumer contract (these are, for example, objects of type `webserver`, `process`, etc.), it can generate and send events to such 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 3.0 enumerated in section 2.1.9; 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 3.19.

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 3.0 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 3.0 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 3.0. In case of script, if `vnx.event(TOPIC, DATA)` is called, the resulting event would have the form of

```
{
    "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 3.0 configuration.

4.2.4 Logging

The development and debugging of scripts in embedded environments is sometimes tough, and Viinex 3.0 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 3.0 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 3.0 log destination (a syslog, or a log file, or a standard error stream), and the `--log-level` policy applied to them.

Note that currently all of that functions accept only one argument, unlike the `console.log()` function in modern browsers. That argument should be either a string (which is logged as is), or an object (which is logged after being encoded into JSON representation).

4.2.5 Linked objects

Probably the most important feature of scripts is that they can control other Viinex 3.0 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 3.0 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.

[2]The same policy is true for the `process` object described in section 2.1.18: an external process can specify the topic and the data of an event, but not the timestamp and origin.

Viinex 3.0 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 3.0 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 3.0 object;
- all methods implemented by all interfaces of corresponding Viinex 3.0 object.

This needs to be explained in more detail. Like with HTTP API, in JS API the case is that one Viinex 3.0 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 implements a given specific interface, – the interfaces in Viinex 3.0 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 3.0 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(....);
```

[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.

– 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++).

4.2.6 Configuration clusters

If the script has its configuration property `clusters` set to `true`, it receives an access to functionality for managing Viinex 3.0 configuration clusters, identical to HTTP API described in section 3.20.

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 3.0 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 3.0 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 2.1.22.

4.2.7 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 3.0 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 3.0 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.

4.3 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 3.0 embedded scripts. Other reason is that the feature of scripting is relatively recent in Viinex 3.0 (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 3.0 team.

The following application interfaces are available for the scripts: `RecControl` (for the recording controller), `LayoutControl` (for the video renderer), `StreamSwitchControl` (for stream switch application), `PtzControl` (for ONVIF devices with PTZ functionality enabled), and `SnapshotSource` (implemented by live video sources, video archives and video analytics modules).

4.3.1 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 2.1.8. Corresponding part of HTTP API is described in section 3.6.

The `RecControl` interface has three methods, `status()`, `record()` 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);
}
```

4.3.2 PtzControl

The `PtzControl` application interface serves for controlling of the PTZ-enabled ONVIF compatible device from scripts in Viinex 3.0.

`PtzControl` has four methods implemented: `gotoPreset`, `gotoHome`, `getPosition` and `moveAbsolute`. This is somewhat limited functionality in comparison with that implemented in corresponding interface in Viinex 3.0 HTTP API which is described in section 3.13, however it is still sufficient for solving a number of applied problems.

The `gotoPreset` and `gotoHome` methods are very similar and serve for the purpose of moving the PTZ device into a predefined position. Method `gotoPreset` accepts one string argument

which should be a name or label of a preset that the camera should be positioned to. Method `gotoHome` does not accept arguments and sends the camera to its "home" position.

The `getPosition` method allows its calling script to obtain the values of current camera's pan, tilt and zoom coordinates. This method does not accept arguments. The result is returned as a tuple of three floating-point numbers which should be interpreted as (pan, tilt, zoom), in that order. Note that the range of returned values is up to the device; with ONVIF devices it is often $[-1, 1]$ for pan, $[0, 1]$ or $[-1, 1]$ for tilt, and $[0, 1]$ for zoom, but these ranges may vary across devices and vendors. Please refer to [12] and section 3.13 of this document for more information on ranges for PTZ coordinates.

The `moveAbsolute` allows the calling script to move the PTZ device into a position with specified absolute coordinates. This method should be called with three arguments, which are expected to be floating point numbers interpreted as the value for pan, tilt and zoom coordinates, in that order. The ranges in which the values should fall are the same as the ranges for values returned by `getPosition` method call. For reference, these ranges can be obtained with the help of "Get PTZ node description" HTTP API call described in section 3.13.1.

4.3.3 LayoutControl

JavaScript interface `LayoutControl` is intended for controlling the layout of a video renderer from scripts in Viinex 3.0. This interface implements two methods, `sources()` and `layout()`, 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 2.4.8.

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.

4.3.4 StreamSwitchControl

The purpose of JavaScript interface `StreamSwitchControl` is to control the stream switch object described in section 2.1.6. This interface exposes two methods, `sources()` and `input()`. 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.

4.3.5 Stateful

The `Statefule` interface exposed via JavaScript is the counterpart of respective HTTP interface described in section 3.18.2. 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 Updateable

The `Updateable` interface is exposed via JavaScript by the same Viinex 3.0 objects which implement the `Updateable` HTTP interface described in section 3.18.2. The purpose of this interface is an abstraction for some object which can accept commands for state modification. The interface has one method `update()` 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.

An example for using the `Updateable` interface can be found in example configuration for railcar identification number recognition system. In particular, that configuration contains the controlling script which uses the `Updateable` inteface implemented by object `ridrcons` in order to start and stop railcar number recognition, and to obtain the recognition results when the recognition for a single railcar is stopped.

4.3.7 SnapshotSource

JavaScript interface `SnapshotSource` is indended for storing the snapshots from an object that provides the `SnapshotSource` interface, on a local filesystem. It exposes one method, `saveFile(destination, source, options)`. This method takes up to three arguments, of which the first – `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 second argument to this method 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 third argument is optional 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).

The `scale` field of the third 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.

To give an example of using this functionality, below is presented an annotated script and configuration file for performing the face detection on a video stream from IP camera, and saving the images of detected faces into a specified folder.

```
// file: save-detected-faces.js

// default configuration -- target folder
// and how much the cropped face image should be extended
var config = {
    target_folder: ".",
    crop_extend_percents: 40
}

var status = {
    saved: 0
}

function onload(cfg){
    if(cfg)
        config = cfg;
    return status;
}

function onevent(e){
    // facial detector sends events with topic "FaceDetection".
    if(e.origin.type != "localvideo" && e.topic != "FaceDetection")
        return;

    // the facial detector in Viinex implements the snapshot source interface
    var src = e.origin.name;
```

```
    if(!vnx.objects[src])
        return;
    var snapshotSource = vnx.objects[src].SnapshotSource;

    // at least it should, bail out otherwise.
    if(!snapshotSource)
        return;

    // construct the path and the file name the snapshot should be saved to
    var ts = e.timestamp.toString().replace(/[ :]/g,'-');
    var target = config.target_folder + "\\" + src + "_" + ts + ".jpg";

    var rect = e.data.face_rect;

    // compute the extended rectangle to be cropped
    if(config.crop_extend_percents){
        var w = rect.right - rect.left;
        var h = rect.bottom - rect.top;

        var dw = w * config.crop_extend_percents / 200.0;
        var dh = h * config.crop_extend_percents / 200.0;

        rect.left -= dw;
        rect.top -= dw;
        rect.right += dw;
        rect.bottom += 2*dw;

        rect.left = Math.max(rect.left, 0);
        rect.top = Math.max(rect.top, 0);
        rect.right = Math.min(rect.right, 1);
        rect.bottom = Math.min(rect.bottom, 1);
    }
    // actually save the snapshot into a file
    var res=snapshotSource.saveFile(target, e.timestamp,
                                    {roi:[rect.left, rect.top,
                                          rect.right, rect.bottom]});
    if(res){
        status.saved += 1;
        vnx.publish(status);
    }
    else vnx.warning("Saving failed");
}
```

This script can be deployed to work with the following configuration file for Viinex 3.0 (note the `init` property of the `savescript` section of configuration):

```
{
    "objects":
    [
        {
            "type": "webserver",
```

```
            "name": "web0",
            "staticpath": "../web/dist",
            "port": 8880
        },
        {
            "type": "onvif",
            "name": "cam1",
            "host": "192.168.0.121",
            "auth": ["admin","12345"]
        },
        {
            "type": "facedet",
            "name": "facedet0",
            "mode": "any",
            "workers": 1,
            "datapath": "share/facedet"
        },
        {
            "type": "facedetvideo",
            "name": "facedetvideo0",
            "freeflow": true,
            "preprocess": 0,
            "postprocess": 0,
            "skip": "while_busy",
            "snapshots": 10
        },
        {
            "type": "script",
            "name": "savescript",
            "load": ["etc/conf.d/save-detected-faces.js"],
            "init": {
                "target_folder": "c:\\temp\\dumpfaces",
                "crop_extend_percents": 50
            }
        }
    ],
    "links":
    [
        [["facedet0","facedetvideo0","cam1"],"web0"],
        [["facedet0","cam1"],"facedetvideo0"],
        ["facedetvideo0", "savescript"]
    ]
}
```

5 Native API

Viinex 3.0 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 3.0. 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 3.0 with access to certain low-level functionality related with video processing, and b) to provide other developers with abilities to extend Viinex 3.0 so suite the needs of their applications.

5.1 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 3.0 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 3.0 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.

5.2 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 3.0 runs; for example this can be an external process object managed by Viinex 3.0 as described in section 2.1.18, 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 3.0 running on a local computer. The underlying methods for acquiring raw video from Viinex 3.0 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 to the

[1]Viinex Virtual Camera is a DirectShow component installed with Viinex 3.0 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 3.0 instance. The video stream rendered by `rend0` object is then sent to each client connected to the virtual camera.

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 3.0 instance. Such shared memory is a scarce resource, and "shared" here means that the instance of Viinex 3.0 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 3.0 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 3.0 instance.

5.3 Implementing the H264 video source plugin

In order to implement the H264 video source plugin for Viinex 3.0, 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 3.0 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 3.0 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 3.0 does not analyze frame or event timestamps generated by plugin, but preserves them. Viinex 3.0 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 3.0 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 Deployment

6.1 Installation

6.1.1 Windows

A Windows Installer database package (MSI) is provided for installing Viinex 3.0 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! When deploying Viinex 3.0 on Windows 7, make sure you have at least Windows 7 Service Pack 1 installed.

NB! Viinex 3.0 only supports installing on 64-bit operating systems.

Viinex 3.0 comes with a number of predefined configuration files, however it is not necessary to use them. It is possible, while installing Viinex 3.0 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 value 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 2.6 is applied. It is possible to choose split configuration when Viinex 3.0 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 3.0 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 3.0 service should be restarted when configuration file or the path to configuration file is changed to reflect the changes.

Note that Viinex 3.0 should not be installed on the same computer where any of previous versions of Viinex software is already installed. To install Viinex 3.0 on such host, one should remove Viinex Foundation 1.4 or Viinex 2.0.

It is also possible to install Viinex 3.0 in unattended mode, which can be useful for including Viinex 3.0 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-3.0.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 3.0 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 3.0 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 3.0 code (see section 6.1.4 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 3.0 when running in Windows service mode.

There are three more optional parameters which can be set for Viinex 3.0 installer from command line when running `msiexec.exe`:

```
SERVICE_NAME,
SERVICE_DISPLAY_NAME,
SERVICE_DESCRIPTION.
```

These parameters allow an application that embeds Viinex 3.0 to set preferrable name, display name and descrition of Viinex 3.0 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 3.0 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.

6.1.2 Linux

For Linux, a Debian package `viinex-3.0.deb` is provided. It can be installed on a Debian-compatible operating system with command

```
dpkg -i viinex-3.0.deb
```

In contrast with Windows installer, deb-package does not contain predefined configuration files, therefor there's nothing to be configured to install Viinex 3.0 on Linux.

Viinex 3.0 is installed on Linux as a System V daemon, registering itself into `init.d` scripts system. However it does not adds itself to any runlevel; the administrator should do so after Viinex 3.0 installation using standard Debian `update-rc.d` utility. The name of Viinex 3.0 service in System V init scripts hierarchy is `viinex`.

By default, Viinex 3.0 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 in the startup script `/etc/init.d/viinex.sh`; corresponding parameters are passed to `viinex` binary as command line arguments. See also the subsection 6.1.3 for more details. 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 2.6 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 3.0 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 3.0 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.

6.1.3 Running Viinex 3.0 in foreground

In both Windows and Linux operating systems, Viinex 3.0 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 3.0 life cycle should be controlled directly by embedding software. For instance, you may decide that starting Viinex 3.0 directly using `CreateProcess` or `fork/exec` is more convenient than treating Viinex 3.0 as NT service or a daemon. For that, Viinex 3.0 executable accepts command-line argument `--foreground`. When this option is given, Viinex 3.0 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 3.0 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 3.0 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 2.6 is applied.

There are also three optional command line arguments: `--log-level=` and `--log-file=`, which can be used to control how and where Viinex 3.0 logs its runtime errors or debug information, and `--threads=` to set the number of OS threads used by Viinex 3.0 to dispatch its execution (see section 6.1.4 for details).

An example command line to start Viinex 3.0 would be

```
viinex --foreground --config=viinex-cfg.json \
       --log-level=INFO --log-file=viinex.log
```

Upon startup in foreground mode, Viinex 3.0 is waiting for an arbitrary data on its standard input before stopping. If Viinex 3.0 is ran manually in a terminal, an "ENTER" key on the keyboard may be hit to stop it. If Viinex 3.0 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 3.0 instance is required to stop.

6.1.4 Setting the number of OS threads

Viinex 3.0 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 3.0 allows for setting the number of OS threads employed for its execution. Viinex 3.0 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 3.0 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 3.0, depending on what objects are specified in Viinex 3.0 configuration. This is the case for image processing tasks like facial detection and vehicle license plate recognition. The `--threads=` setting does not affect these dedicated threads.

The need for limiting the number of OS threads used by Viinex 3.0 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 3.0 process[1]

[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 3.0 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.

6.2 License key management

For license keys management, `viinex-lm-upgrade` utility is included in Viinex 3.0 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, and
- 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'.

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.

6.2.1 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
```

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 concrete dongle, which the issued command should be applied to.

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
  LPRecognizer                       4
  LPRecognizer_Viinex                4
  LPRecognizer_Static10              4
====================================
```

6.2.2 Obtaining information on PC hardware

As mentioned in section 2.5, 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 3.0, 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 3.0 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 3.0 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 3.0 licensor will provide you with a license document suitable for using in the `license` part of the configuration (see section 2.5).

6.2.3 Upgrading a 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.

6.2.4 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