vdSM / vDC API

Contents

Basics

- based on JSON-RPC 2.0, see http://www.jsonrpc.org/specification
- Specifications of methods and notifications in this documentation do not list JSON-RPC 2.0 level parameters like *jsonrpc* and *id*, their presence is implied by declaring the communication as JSON-RPC 2.0 herewith.
- transport level is a TCP socket connection, established by the vDC to the vdSM, and kept open as long as possible.
- In the stream of JSON request and answer objects sent over the connection, objects must be separated by a LF character.
- The life time of the connection defines the vDC session. If the connection breaks, a new session needs to be established.
- Design goal for the API was to reduce the number of methods as much as possible (compared to the vdSM preliminiary device API). Only the most basic actions common to all dS devices (*callScene, saveScene, ping, pong,* ...) are implemented as separate methods, everything else should be handled via reading and writing named properties (eg. functionality like former *SetLocalPrio, ProgModeOn, SetOutVal…*).

Discovery

Requirements for vDCs

- vDC needs to be able to be discovered by vdSMs (embedded within a dS installation) automatically on a given network.
- It must be avoided that a vdSM connects to a wrong (neighbour's, for example) vDC
- It must be avoided requiring vDCs to have UI of their own.

Requirements for vdSMs

The same requirements exist for vdSMs - these need to be discoverable by dss/dsa/ds485p such that they can be automatically connected.

Short term solution

- vDCs and vdSMs announce their services using Avahi (gnu implementation of Apple's Bonjour)
- vdSMs look for available vDCs in the Avahi announcements and connect to at least one of them. vDCs might reject connections if they are already connected to another vdSM.
- dss/dsa/ds485p look for available vdSMs in the Avahi announcements and connect them.
- vDCs that come bundled with a vdSM in the same physical device (as it is planned for plan44.ch DALI/enOcean bridge) may choose to not announce the vDC when the in-device vdSM is configured to always connect the in-device vDC.
- likewise, vdSMs that come bundled with a dss/dsa/ds485p might be directly tied to their in-device dss/dsa/ds485p without announcing it into the LAN.

- To avoid wrong connections, the instance that initiates a connection must be able to check possible peers received via Avahi against an optional whitelist. If the whitelist exists, only listed peers might be connected. The idea is that in small/simple installations connection is fully automatic, but if conflicts arise in large setups (like aizo dev space) these can be solved by adding whitelists.

Long term solution idea

The whitelist might get replaced or supplemented by more end-user friendly mechanism, possibly similar to how dss API clients are approved now (client requests and receives a token, for which the user can grant access in the web configurator)

Avahi service description files

- On a system with avahi-daemon installed, announcing services consists of creating .service files and putting them into /etc/avahi/services
- The service types are chosen to be very unlikely to collide with other company's services by using the "ds-" prefix. If needed, dS service names could still be registered with IANA later (see http://www.rfc-editor.org/rfc/rfc6335.txt)
- The port numbers used below are just examples, actual ports might differ.
- Once dS services can handle ipv6, the "protocol" attribute should be set to "any"

/etc/avahi/services/ds-vdc.service

```
<?xml version="1.0" standalone='no'?>
<!DOCTYPE service-group SYSTEM "avahi-service.dtd">
<service-group>
  <name replace-wildcards="yes">digitalSTROM vDC on %h</name>
  <service protocol="ipv4">
    <type>_ds-vdc._tcp</type>
    <port>8444</port>
  </service>
</service-group>
```

/etc/avahi/services/ds-vdsm.service

```
<?xml version="1.0" standalone='no'?>
<!DOCTYPE service-group SYSTEM "avahi-service.dtd">
<service-group>
  <name replace-wildcards="yes">digitalSTROM vdSM on %h</name>
 <service protocol="ipv4">
    <type>_ds-vdsm._tcp</type>
    <port>8441</port>
  </service>
</service-group>
```

Error Codes

- JSON-RPC 2.0 standard error codes (-32768..-32000) are used for syntax errors on the JSON-RPC 2.0 level.

- Errors specific to this API are aligned with the well known HTTP error codes as much as possible for easier understanding (see http://en.wikipedia.org/wiki/List_of_HTTP_status_codes)

vDC Session

Basics

- a session is not identical with having a TCP connection. Using the *pushURI/ allowDisconnect* mechanism, vdSMs may choose to create a connection only when needed and stay disconnected during idle time.
- a vdSM aims to keep vDC sessions active all the time.
- if a session is terminated for any reason, the vdSM must try to establish a new session.
- Only if fundamental incompatibility between vDC and vdSM is detected (no common API version), the vdSM might cease trying to establish a session..
- at a given time, a vdSM might have at most one single session with one particular vDCs (it may of course have connections to multiple distinct vDCs)
- a vDC session must always start with a session initialisation phase, before it changes into the operation phase.

vDC Session initialisation

- vdSM connects to the vDC
- vdSM calls "Hello" method on vDC.

Method	*hello*	vdSM -> vDC
Request Parameter	JSON type	Description
dSID	string	dSID of the vdSM
APIVersion	string	vDC API version At the writing of this specification *APIVersion* is "1.0"
pushURI	optional string in tcp://host:port format	host/port the vDC can use to connect to the vdSM in case the vdSM has disconnected during idle time and events need to be delivered upstreams.
Response		
result (on success)	object	
dSID	string	dSID of the vDC Note: although the vDC does not yet appear as a dS device in the current digitalSTROM specification, it might have in the future so it must have a dsid as a globally unique identifier.
allowDisconnect	bool	only if the vDC returns *true*, the vdSM is allowed to disconnect from the vDC without terminating the vDC session. If the *Hello* call does not contain a *pushURI,* the vDC must always return *false* here. The vDC may also return *false* if *pushURI* was present in case it does not implement the pushURI mechanism.
error (on failure)	Error Object	503 : service not available: This means the vDC cannot accept the connection because it is already connected to *another* vdSM. Note that vDCs must NOT reject reconnection from an already connected vdSM, but must start a new vDC session. 505 : incompatible API version. The vdSM does not support the API version presented in *APIVersion*

vDC Session operation

- session operation consists of establishing none, one or multiple Device Sessions (see below)
- To avoid a vDC session to implicitly end, some minimal communication must occur between vdSM and vDC in regular intervals (for example: *ping/pong*)

vDC Session termination

- a vDC session is explicitly terminated when the *Bye* command is called:

Method	*bye*	vdSM -> vDC
Request Parameter	JSON type	Description
Response		
result (on success)	NULL	after receiving *Bye* vDC closes connection.

- a vDC session is implicitly terminated when a *Hello* command asks for starting a new vDC session
- if the current vDC session was established with *Hello* returning *allowDisconnect=false*, closing the connection implicitly terminates the vDC session
- if the current vDC session was established with *Hello* returning *allowDisconnect=true*, the vDC session is kept alive even when the connection is closed, but will implicitly end after some period of no activity (timeout).
- when the vdSM disconnects a session that was established with *allowDisconnect=true*, it can reconnect and continue the session. The vDC must check the originator (IP address) to match the originiator of the initial *Hello* command when a vdSM reconnects to ensure session integrity.
- if a vDC push (see below) fails, the vDC may also implicitly terminate the vDC session (usually only after a few retries)
- ending the vDC session implicitly terminates all device sessions.

vDC push connection

- When the vDC has an active vDC session, but the vdSM does not currently have a connection open to the vDC, and there is an event (*pushProperty, Announce, Vanish*) to report upstreams, the vDC must actively connect the vdSM at the *pushURI* established in the *Hello* command, deliver one or multiple event notification(s) and close the connection.
- The vDC may not keep this connection open without actually sending data or awaiting immediate method call results. Waiting for results must have a short timeout after which the connection is closed even if no result arrives.

Device Session

- after vDC session is established, the vDC must start a device session for all of the devices managed by the vDC.
- A device is considered managed by the vDC when the vDC has resonably reliable information that the device is in fact connected to or connectable from the vDC. This means that the vDC should start a device session even for temporarily offline devices.
- The vdSM can explicitly request removal of managed devices

Device Session Initialisation

- vDC calls "Announce" method on vdSM

vDC Method	*announce*	vdSM -> vDC
Request Parameter	JSON type	Description
dSID	string	The dSID of the device to be announced
Response		
result (on success)	NULL	
error (on failure)	Error Object	507 : insufficient storage - vdSM cannot handle another device

Device Session Operation

- during a device session, device level methods can be invoked by either party on the other, and device level notification can be sent by either party to the other.
- See separate chapter on Device Level Method Notifications below for specification of individual methods/notifications supported

Device Session Termination

- either vDC sends "Vanish" notification to vdSM to indicate a device has been physically disconnected or unlearned (think: enOcean unidirectional switches for example) from the vDC.

Notification	*vanish*	vDC -> vdSM
Notification Parameter	JSON type	Description
dSID	string	The dSID of the device that has vanished

- or vdSM calls "Remove" method on vDC to request a device to be removed from this vDC (but might have been connected to another vDC in the meantime). vDC may reject removal only if it has 100% knowledge the device is actually connected and operable (in which case the higher levels in dS should see the device as active anyway and will not allow users to delete it)

vDC Method	*remove*	vdSM -> vDC
Request Parameter	JSON type	Description
dSID	string	The dSID of the device to be removed
Response		
result (on success)	NULL	
error (on failure)	Error Object	403 : forbidden - vDC does not allow to remove the device, for example because it is verifiably physically connected to the vDC. However, consider that for example wireless devices might get carried away without being un-paired from their former vDC, and then paired with another vDC in the same installation. In such a case, dss/dsa will want to remove the device from the former vDC/vdSM - the vDC must not reject such a deletion attempt.

Device and vDC Operation Methods and Notifications

- Note: the vDC itself has a dSID and can be addressed by some (not all) of the device level methods, in particular reading and writing named properties for vDC wide configuration and the presence polling method *ping*.

- All device level methods share the following structure and generic error conditions, which are not listed again with each method description below, but their presence is implied by declaring all methods in this chapter as vDC Operation methods herewith.

Method name	*<see device level methods below>*	
Request Parameter	JSON type	Description
dSID	string	The dSID of the device the method relates to (when coming from vdSM: the target device, when coming from vDC: the originating device)
Response		
error (on failure)	Error Object	401 : unautorized - this is returned when there is no vDC session in operation mode. The caller must start a new vDC session using the *Hello* method before issuing vDC Operation Methods. 404 : not found - the receiver (device or vDC itself) specified through dSID is unknown at the callee.

- All device level notifications share the following structure, which are not listed again with each method description below, but their presence is implied by declaring all notifications in this chapter as device level notifications herewith.

Notification name	*<see device level notifications below>*	
Notification Parameter	JSON type	Description
dSID	string	The dSID of the device the notification relates to (when coming from vdSM: the target device, when coming from vDC: the originating device)

Named Property access

- devices have named properties which can be read and written by the vdSM and in some cases being pushed from the vDC.
- supported value types for properties are basic JSON types integer, double, boolean and string, or a JSON object consisting of multiple sub-fields of these types.
- Properties with JSON object types can only be read as a whole, and are mainly intended to structure data that would otherwise be encoded in bitfields of an integer, like sensor table entries etc.
- When writing properties with JSON object types, object fields that are not included will not be changed (NOT set to a default!). This way, it is possible to change single values in structured properties.
- some properties might exist in multiple instances (array elements). There are four ways to access an array:
  - A single element can be adressed for reading and writing by using the *index* parameter (0..n)
  - The size of the array can be queried by accessing the array with *index* set to -1
  - A range of elements can be addressed for read (but not for write) using the *offset* parameter specifying the index of the first element to return, plus an optional *count* parameter specifying the number of elements to return. Omitting the *count* parameter means "to the end of the array". In both cases, the vDC might return less elements

than requested due to memory constraints. So to read an entire array, the size of the result should be compared with the total size of the array (as returned in *index* -1) and more requests issued until all elements are returned. Alternatively, requests with increasing *offset* can be issued until error status 204 is returned.
    - Accessing the array for read without *index/offset/count* parameters is equivalent with an access specifying *offset*=0 (i.e. returns the entire array or the beginning of the array in case of vDC memory constraints).

- The available properties depend on the type of device - the complete set of properties supported by a device comprises the *device profile*
- A common set of properties called *common device properties* (see below) must be supported by all devices. These properties can be read to identify a device and its functionality and associate it with a specific *device profile*.
- Compared to R105 digitalStrom devices, named properties replace (or encapsulate) device parameters organized with bank/offset addressing scheme.
- the vDC itself can have named properties that can be addressed via its dSID and the methods described below.

Reading a Property value

Method name	*getProperty*	vdSM -> vDC
Request Parameter	JSON type	Description
name	string	The name of the property
index	optional integer	For array properties only: If >=0: the index of a single array element to return. If == -1: request that the size of the array is to be returned as an integer number
offset	optional integer>=0	For array properties only: Specifies the index of the first element to be returned
count	optional integer>=1	For array properties only: Specifies the number of elements to be returned starting from *offset.* The vDC may decide to return less elements due to memory constraints. Omitting count from a *getProperty* call which specifies *offset* means "all elements to the end of the array".
Response		
result (on success)	depends on actual property: integer, double, string, boolean, object, NULL, or array containing elements of one of the types above.	• for non-array properties and single elements of array properties addressed by *index*, NULL to signal "no value" (but property exists) • for array properties accessed with *offset* and *count* (or no params at all implying *offset*=0 and *count*=max): a JSON array containing array elements up to the maximum count specified. • for array properties when accessed with *index==-1:* the size of the array as an integer number
error (on failure)	Error Object	501 : not implemented - this device does not implement this property 204 : no content for this array element. This code is returned when the *index or offset* parameter address array elements which do not exist 403 : forbidden - the property exists but cannot be read (write-only, uncommon case).

Writing a property value

Method name	*setProperty*	vdSM -> vDC
Request Parameter	JSON type	Description
name	string	The name of the property
index	optional integer	For array properties only: if >=0: the index of the array element to write. if == -1: for arrays with dynamic size, the *value* indicates the new size of the array. For array with fixed size, trying to write *index* -1 will return Error 403.
value	depends on actual property: integer, double, string, boolean, object, NULL, or array containing elements of one of the types above.	• for non-array properties and single elements of array properties addressed by *index*, NULL to signal "no value" (but property exists)
Response		
result (on success)	NULL	
error (on failure)	Error Object	501 : not implemented - this device does not implement this property 204 : no content for this array element. This code is returned when the *index* parameter addresses a array element which does not exist 415 : invalid value type (for writing) 403 : forbidden - the property exists but cannot be written (is read-only, or is an array and *index* is missing, or *index* is -1 and the array does not have a dynamic size).

Getting notified of a property value (change)

- Some properties (especially button/input/sensor states) might change within the device and can be reported to the dS system via pushProperty, avoiding the need for the vdSM to poll values.

Notification name	*pushProperty*	vDC -> vdSM
Notification Parameter	JSON type	Description
name	string	The name of the property
index	optional integer>=0	if present, the pushed property value is an element in an array property. *index* indicates which array element is being pushed
value	depends on actual property, object, integer, double, string, boolean or NULL	integer or string value of the property (element), depending on property, NULL to signal "no value"

Presence polling

- The presence polling is available for every device, and also for the vDC itself. Implementation should return a pong only if the device can be considered active in the system. If possible at reasonable cost, a connection test with the device's hardware should be made. In some cases (unidirectional sensors) it might not be possible to query the device, in these cases the vDC should apply reasonable heuristics to decide whether to report the device as active or not.

Notification name	*ping*	vdSM -> vDC
Notification Parameter	JSON type	Description

Notification name	*pong*	vDC -> vdSM
Notification Parameter	JSON type	Description

Actions

Call Scene

- calls a scene on the device

Notification name	*callScene*	vdSM -> vDC
Notification Parameter	JSON type	Description
scene	integer	dS scene number to call
force	boolean	if true, local priority is overridden
group	optional integer	dS group number, present if the scene call was not applied to a single device, but a zone/group (informational only, vdSM already creates separate calls for every involved device)
zoneID	optional integer	dS global zone ID number, present if the scene call was not applied to a single device, but a zone/group (informational only, vdSM already creates separate calls for every involved device)

Save Scene

- save the relevant parts of the current state of the device (usually the output value, but possibly multiple output values, flags, etc. in future devices) as scene.

Notification name	*saveScene*	vdSM -> vDC
Notification Parameter	JSON type	Description
scene	integer	dS scene number to save state into
group	optional integer	dS group number, present if the scene call was not applied to a single device, but a zone/group (informational only, vdSM already creates separate calls for every involved device)
zoneID	optional integer	dS global zone ID number, present if the scene call was not applied to a single device, but a zone/group (informational only, vdSM already creates separate calls for every involved device)

Undo Scene

- undoes the last scene call.

Notification name	*undoScene*	vdSM -> vDC
Notification Parameter	JSON type	Description
currentScene	optional integer	if present, this specifies the current scene. Undo is executed only if device's scene matches currentScene, or currentScene is not specified
group	optional integer	dS group number, present if the scene call was not applied to a single device, but a zone/group (informational only, vdSM already creates separate calls for every involved device)
zoneID	optional integer	dS global zone ID number, present if the scene call was not applied to a single device, but a zone/group (informational only, vdSM already creates separate calls for every involved device)

Identify

- identify the device for the user - usually implemented as blinking the controlled light or an indicator LED the device might have. Depending on device type, the alert might be implemented differently, such as a beep, or hum or short movement.

Notification name	*identify*	vdSM -> vDC
Notification Parameter	JSON type	Description
group	optional integer	dS group number, present if the scene call was not applied to a single device, but a zone/group (informational only, vdSM already creates separate calls for every involved device)
zoneID	optional integer	dS global zone ID number, present if the scene call was not applied to a single device, but a zone/group (informational only, vdSM already creates separate calls for every involved device)

Common properties

- The common properties must be supported by entities which can be addressed via a dSID using this API. At the time of writing, this includes virtual devices and the vDC itself, but may be extended to other entities.
- The "type" property reveals the kind of entity.
- Properties common to a specific entity type are maintained in separate documents. In particular, see "vdSD properties" document for common properties of virtual devices.

property name	acc	Type/range	description	R105 mapping
dSID	r	string	the dSID of the entity. Normally not used in regular vDC API calls, as these require the dSID in the getProperty() call. Useful for debugging.	
idBlockSize	r	integer	The number of consecutive dSIDs (starting with this device's dSID) that are reserved. This means that the next *idBlockSize*-1 dSIDs are guaranteed not being used by any vdSD in any vDC. This mechanism allows that a vdSM can implement representing multi-input devices as multiple dSIDs for backwards compatibility with dS 1.0	
type	r	string	Type of entity: "dSD" : real dS device block "vdSD" : virtual dS device "vDC" : the vDC itself "dSM" : a real dSM "vdSM" : a vdSM "dSS" : a dSS "*" : unknown/internal/other	
model	r	string	human-readable model string of the entity. Should be descriptive enough to allow a human to associate it with a piece of hardware or software	ProductID Bank 1 - 0x0A
hardwareVersion	r	string or NULL	string describing the hardware device's version represented by this entity, if available	hardware

property name	acc	Type/range	description	R105 mapping
hardwareGuid	r	string or NULL	hardware's product and item code, if any, in URN format such that not only the (preferred) SGTIN96 can be specified. See http://www.epc-rfid.info/sgtin This ID can be used to identify multiple devices that all belong to a single hardware unit, see *numDevicesInHW* and *deviceIndexInHW.*	dsid equals this or derives from this
oemGuid	r	string or NULL	product and item code of the product the hardware is embedded in, if any, in URN format such that not only the (preferred) SGTIN96 can be specified. See http://www.epc-rfid.info/sgtin	Bank 3 - 0x2a..0x2f Bank 1 - 0x1c..0x1d
name	r/w	string	user-specified name of the entity. Is also stored upstreams in the vdSM and further up, but is useful for the vDC to know for configuration and debugging. The vDC usually generates descriptive default names for newly connected devices. When the vdSM registers a device, it should read this property and propagate the name towards the dSS. When the user changes the name via the dSS configurator, this property should be updated with the new name.	