vdSM / vDC API

Contents

Basics

- based on Google protocol buffers, see https://developers.google.com/protocol-buffers
- ~~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 vdSM to the vDC. Usually kept open over a longer period to exchange multiple requests/answers.
- ~~In the stream of JSON request and answer objects sent over the connection, objects must be separated by a LF character.~~
- The TCP stream consists of a 2-byte header containing the message length (16 bits, in network byte order, maximum accepted length is 16384 bytes) followed by the protocol buffer message.
- The life time of the connection defines the vDC session. If the connection breaks, a new session needs to be established. (There is however a mechanism that allows the vdSM to close and re-open the connection to save network resources in situations when keeping the connection open is not feasible)
- 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 *ProgModeOn, SetOutVal…*).

Terms

(logical) vDC	virtual device connector. A vDC is primarily a logical entity within the dS system and has its own dSUID. A vDC represents a type or class of external devices.
vDC host	network device offering a server socket for vdsm to connect to. One vDC host can host multiple logical vDCs, if the host supports multiple device classes.
vdSM	virtual digitalSTROM Meter. A vdSM can connect to one or several vDC hosts to connect one or several logical vDCs to the dS system.
vdSD	virtual digtalSTROM device. A vdSD represents a single device in the dS system, and behaves like a real digitalSTROM terminal block (dSD, digitalSTROM device)
vDC session	logical connection between a vdSM and a vDC host (representing one or multiple vDCs)

Discovery

Requirements for vDC hosts

- vDC hosts 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 hosts
- 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

- vDC hosts and vdSMs announce their services using Avahi (gnu implementation of Apple's Bonjour)
- vdSMs look for available vDC hosts in the Avahi announcements and connect to at least one of them. vDC hosts 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.
- vDC hosts 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 host when the in-device vdSM is configured to always connect the in-device vDC host.
- 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 host 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>
```

•

vDC host Session

Basics

- a session represents the connection from a single vdSM to a single vDC host (which may host one or multiple logical vDCs)
- 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 host 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 vDC host (it may of course have connections to multiple distinct vDC hosts)
- a vDC session must always start with a session initialisation phase, before it changes into the operation phase.

vDC host Session initialisation

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

Method	*hello*	vdSM -> vDC host
Request Parameter	Type	Description
dSUID	string	dSUID 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	
dSUID	string	dSUID of the vDC host Note: although the vDC host does not yet appear as a logical entity in the current digitalSTROM specification, it still has a dSUID in order to be addressable by custom apps and for future dS system evolution
allowDisconnect	bool	only if the vDC host returns *true*, the vdSM is allowed to disconnect from the vDC host without terminating the vDC session. If the *Hello* call does not contain a *pushURI,* the vDC host must always return *false* here. The vDC host may also return *false* if *pushURI* was present in case it does not implement the pushURI mechanism.
error (on failure)	Error Object	RESULT_SERVICE_NOT_AVAILABLE: This means the vDC host cannot accept the connection because it is already connected to *another* vdSM. Note that vDC hosts must NOT reject reconnection from an already connected vdSM, but must start a new vDC session. RESULT_INCOMPATIBLE_API: The vdSM does not support the API version presented in *APIVersion*

vDC host Session operation

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

vDC host Session termination

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

Method	*bye*	vdSM -> vDC host
Request Parameter	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 host 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 host may also implicitly terminate the vDC session (usually only after a few retries)

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.

vDC Announcement

- after vDC session is established, the vDC host must announce every logical vDC it hosts, before it announces any of that logical vDC's devices.
- It does so by calling the "announcevdc" method
- unlike individual devices (see below), logical vDCs cannot vanish during an established vDC session.

vDC Method	*announcevdc*	vDC host -> vdSM
Request Parameter	Type	Description
vdcdSUID	string	The dSUID of the vDC to be announced
Response		
result (on success)	NULL	
error (on failure)	Error Object	RESULT_INSUFFICIENT_STORAGE - vdSM cannot handle another vDC

vdSD Announcement

- for every vDC announced, the vDC must announce all device 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 announce devices even if the device is temporarily offline.
- The vdSM can explicitly request removal of managed devices

Device Announcement

- vDC calls "Announce" method on vdSM

vDC Method	*announce*	vDC host -> vdSM
Request Parameter	Type	Description
dSUID	string	The dSUID of the device to be announced
vdcdSUID	string	The dSUID of the logical vdc which contains this device
Response		
result (on success)	NULL	
error (on failure)	Error Object	RESULT_INSUFFICIENT_STORAGE - vdSM cannot handle another device

Device Operation

- after announcing a device, device level methods can be invoked by either party on the other, and device level notifications 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

Ending Device Operation

- 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 host -> vdSM
Notification Parameter	Type	Description
dSUID	string	The dSUID 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 host -> vDC
Request Parameter	Type	Description
dSUID	string	The dSUID of the device to be removed
Response		
result (on success)	NULL	
error (on failure)	Error Object	RESULT_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 host and every logical vDC have dSUIDs and can be addressed by some (not all) of the device level methods, in particular reading and writing named properties for vDC 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	Type	Description
dSUID	string	The dSUID of the entity (device or vDC) the method relates to (when coming from vdSM: the target entity, when coming from vDC: the originating entity)
Response		
error (on failure)	Error Object	RESULT_NOT_AUTHORIZED - 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. RESULT_NOT_FOUND - the receiver (device or vDC itself) specified through dSUID 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	Type	Description
dSUID	string	The dSUID of the entity (device or vDC) the notification relates to (when coming from vdSM: the target entity, when coming from vDC: the originating entity)

Named Property access

- *Adressable entities* are items within the vdc host that have their own dSUID that can be specified in vDC API request to specifically address them. The vDC host as a whole, the contained vDC(s) and every virtual device have their own dSUID and thus are addressable entities.
- Adressable entities have named properties which can be read and written by the vdSM and in some cases being pushed from the vDC.
- Properties that are defined in the digitalSTROM specifications (including this document) with name, type and behaviour are considered *system properties*. Implementations conforming to the specification must support these.
- Additionally, implementations might add *implementation specific properties* to extend functionality beyond what the dS system demands. These properties' names must always be prefixed by "x-". It is further recommended to include an identifier for the party who introduces a property. For example company Abc Inc. could prefix their properties with "x-abc-".
- Supported value types for properties are the simple types integer, double, boolean and string, or a structured object type containing multiple named sub-fields. Only one level of sub-fields is supported, i.e. all sub-fields are always simple types.
- Properties of type object can only be read as a whole (all available sub-fields are returned).
- When writing properties with object types, sub-fields that are not included in the value written 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
  - Multiple consecutive elements of an array can be read with a single getProperty request by specifying a *count* parameter > 1. The count parameter specifies the maximum number of elements to return, a vdc implementation 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.
- The available properties depend on the kind of the addressable entity (vdc, vdc host) - the complete set of properties supported by a virtual device entity is defined in the *device profile* for that type of device.
- A common set of properties called *common properties* (see below) must be supported by all adressable entities. 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 dSUID and the methods described below.

Reading a Property value

Method name	*getProperty*	vdSM -> vDC host
Request Parameter	Type	Description
name	string	The name of the property
index	optional integer	Defaults to 0 For array properties, this allows accessing individual elements. For non-array properties, only index==0 returns something. If index is specified -1, the size of the array is returned as an integer number. Non-array properties always return 1.
count	optional integer>=1	Defaults to 1 For array properties only: Specifies the number of elements to be returned starting from *index.* The vDC may decide to return less elements due to memory constraints.
Response		
result (on success)	depends on actual property: integer, double, string, boolean, object, NULL, possibly repeated when array property is queried with count>1	• NULL value to signal "no value" (but property/array element exists) • for any property when accessed with *index==-1:* the number of elements (1 for non-arrays, number of elements for arrays) as an integer number
error (on failure)	Error Object	RESULT_NOT_IMPLEMENTED - this entity does not implement this property RESULT_NO_CONTENT_FOR_ARRAY - This code is returned when the *index or offset* parameter address array elements which do not exist RESULT_FORBIDDEN - the property exists but cannot be read (write-only, uncommon case).

Writing a property value

Method name	*setProperty*	vdSM -> vDC host
Request Parameter	Type	Description
name	string	The name of the property
index	optional integer	Defaults to 0 For array properties, this allows accessing individual elements. For non-array properties, only index==0 can be written.
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). • for batch array writes (*index* and *count>1* specified, see below), the value to be set equally in all specified array elements. • for array properties when accessed with *index==-1:* the new size of the dynamic array as an integer number. Arrays with static size will return error 403
count	optional integer>=1	For array properties only: Specifies the number of elements to be written starting from *index.*
Response		
result (on success)	NULL	
error (on failure)	Error Object	RESULT_NOT_IMPLEMENTED - this device does not implement this property RESULT_NO_CONTENT_FOR_ARRAY - This code is returned when the *index* parameter addresses a array element which does not exist, or not all elements in the given *offset/count* range for a batch write exist (note however that some elements might already be written despite this error) RESULT_INVALID_VALUE_TYPE - passed type is wrong and cannot be written. RESULT_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 host -> vdSM
Notification Parameter	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 addressable entity (vDC host, vDCs and all devices). Implementation should return a pong only if the entity 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 host
Notification Parameter	Type	Description

Notification name	*pong*	vDC host -> vdSM
Notification Parameter	Type	Description

Actions

Actions are operations that may change the internal state of the device and/or its outputs, often depending on preconditions, but do not cause a distinct change of a single status value that could be read back.
Distinct, unconditional state changes that can be read back are always implemented as properties, not actions.

Call Scene

- calls a scene on the device

Notification name	*callScene*	vdSM -> vDC host
Notification Parameter	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 host
Notification Parameter	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 a scene call. All output values are restored to the state they had before the scene call.

Notification name	*undoScene*	vdSM -> vDC host
Notification Parameter	Type	Description
scene	integer	This specifies the scene call to undo. Undo is executed only if device's last called scene matches *scene*. This is to prevent undoing a scene which might have been called in the meantime from another origin (like a local on or off).
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)

Set local priority

- Sets the device into local priority mode (i.e. sets the *localPriority* property) if the passed scene does not have the *dontCare* flag set. This is used for including devices into area operations. Note that this is a compatibility method to simplify dS 1.0 interfacing and might be removed later in dS 2.x.

Notification name	*setLocalPriority*	vdSM -> vDC host
Notification Parameter	Type	Description
scene	integer	This specifies the scene to check for the *dontCare* flag. If it is set, nothing happens, if it is not set, the localPriority flag will be set on the device level.
group	optional integer	dS group number, present if local priority was not set for a single device only, 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 local priority was not set for a single device only, but a zone/group (informational only, vdSM already creates separate calls for every involved device)

Call Minimum Scene

- if the device is off, set it to the minimal value needed to become logically switched on and participate in dimming. Otherwise, no action is taken.

Notification name	*callSceneMin*	vdSM -> vDC host
Notification Parameter	Type	Description
scene	integer	This specifies the scene to check for the *dontCare* flag. If it is set, nothing happens, if it is not set, and the device is off, the minimum output level will be set.
group	optional integer	dS group number, present if the call was not applied to a single device only, 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 call was not applied to a single device only, 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 host
Notification Parameter	Type	Description
group	optional integer	dS group number, present if the identify notification was not sent 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 identify notification was not sent to a single device, but a zone/group (informational only, vdSM already creates separate calls for every involved device)

setControlValue

- sets a control value (a dS "sensor value" coming in from a dS sensor or generated by a dSS app such as a heating regulator) to the device, which will evaluate it according to the device's group (color) and settings, and derive actual output values from it (for example a valve position).

Notification name	*setControlValue*	vdSM -> vDC host
Notification Parameter	Type	Description
name	string	The name of the control value. This defines the semantic meaning of the value and thus how the value will affect (or not) the output(s) of the device. In dS 1.0, control name values are mapped to dS "sensor type" (see dS sensor type table) numbers. Currently defined mappings: • "heatingLevel" (dS Sensortype 50): level of heating intensity -100..100: 0=no heating or cooling, 100=max heating, -100 max cooling
value	double	control value (aka dS sensor value)
group	optional integer	dS group number, present if the control value was not sent to a single device only, but to 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 control value was not sent to a single device only, but to 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 dSUID using this API (*addressable entities*). At the time of writing, this includes virtual devices, logical vDCs and the vDC host, 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
dSUID	r	string	the dSUID of the entity. Normally not used in regular vDC API calls, as these require the dSUID in the getProperty() call. Useful for debugging.

property name	acc	Type/range	description
idBlockSize	r	integer	The number of consecutive dSUIDs (starting with this device's dSUID) that are reserved. This means that the next *idBlockSize*-1 dSUIDs 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 dSUIDs for backwards compatibility with dS 1.0
type	r	string	Type of entity: "vdSD" : virtual dS device "vDC" : a logical vDC "vDChost" : the vDC host "vdSM" : a vdSM
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
hardwareVersion	r	string or NULL	string describing the hardware device's version represented by this entity, if available
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.*
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
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.