亚马逊云科技

Extension SDK Developer Guide

NICE DCV

NICE DCV: Extension SDK Developer Guide

Table of Contents

What is the NICE DCV Extension SDK?

NICE DCV is a high-performance remote display protocol. It lets you securely deliver remote desktops and application streaming from any cloud or data center to any device, over varying network conditions. By using NICE DCV with Amazon EC2, you can run graphics-intensive applications remotely on Amazon EC2 instances. You can then stream the results to more modest client machines, which eliminates the need for expensive dedicated workstations.

With the NICE DCV Extension SDK, developers can integrate NICE DCV protocol with their applications. The following are typical use cases:

- Provide high-level device redirection for custom hardware devices in remote sessions.
- Establish virtual channels between NICE DCV Server and NICE DCV Client to enhance remote application usability.
- Describe the NICE DCV client and NICE DCV server runtime components and allow applications to interact with them.

A DCV extension may communicate with either a NICE DCV client or a NICE DCV server, depending on where it is installed. In addition, the NICE DCV Extension SDK could request a virtual channel via the NICE DCV protocol and then use this virtual channel to send arbitrary data.

The purpose of this guide is to explain how to integrate third-party applications into the NICE DCV Protocol using the NICE DCV Extension SDK (Software Development Kit) and its associated API (Application Programming Interface).

Topics

Prerequisites

Before you start working with the NICE DCV Extension SDK, ensure that you're familiar with NICE DCV and NICE DCV sessions. For more information, see the NICE DCV Administrator Guide.

The NICE DCV Extension SDK supports NICE DCV server and NICE DCV Extension SDK version 2023.0 and later.

NICE DCV Extensions are not supported with a Web Client

API Categories

There are three categories of NICE DCV Extension SDK APIs:

- **General API -**Available on both client and server, this API provides messages describing the NICE DCV components and service information.
- **Virtual Channel API -**Allows extensions to create virtual channels for transferring data between client and server. API is available on both client and server, and its behavior is symmetrical. Extensions running on either side can establish virtual channels.
- **Geometry API -**Available only for client-side extensions, this API provides messages to obtain local views and the layout of the remote desktop.

Versioning convention

The NICE DCV Extension SDK version is defined in the following format: `*major*.*minor*.*patch*`. The versioning convention generally adheres to the semantic versioning model. A change in the major version, such as from `1.x.x` to `2.x.x`, indicates that breaking changes that might require code changes and a planned deployment have been introduced. A change in the minor version, such as from `1.1.x` to `1.2.x`, is backwards compatible, but might include deprecated elements.

Getting started with the NICE DCV Extension SDK

The NICE DCV Extension SDK uses Protocol Buffers (protobuf), an open-source data format designed to serialize structured data in a compact, binary form. By using Protocol Buffers, NICE DCV components and extensions can communicate with one another in a way that is platform-independent and extensible, regardless of the selected programming language.

The definitions of the messages are contained in the file `extensions.proto`.

To get started with the NICE DCV Extension SDK

1. Get familiar with Protocol Buffers (protobuf)
2. Review the code samples available at https://github.com/aws-samples/dcv-extension-sdk-samples
3. Download the protocol definition file `extensions.proto` from the Github repository at https://github.com/aws-samples/dcv-extension-sdk

Topics

Extension architecture

There are two parts to the NICE DCV Extension SDK: a `manifest` file that describes extension metadata and an `executable` that runs when the extension is activated.

There is no difference in the structure of client and server extensions.

Extension manifest

Manifest files are JSON files in the format described below. Ensure that all special characters are properly escaped with a reverse backslash. A `.json` extension is required for the manifest file.

```
{
"name": "MyExtension",
"description": "My extension longer description",
"path": "C:\\Program Files\\My Company\\My Product\\bin\\myextension.exe",
"start_on_server": true,
"start_on_client": true,
"virtual_channel_namespace": "com.company.product",
"userdata": "parameters to the extension"
}
```

- **name –** The name for the extension that will appear in DCV logs.
- **description –** A textual description of the extension that appears in the user interface (About dialog box).
- **path –** The complete path to the extension executable, escaping backslashes if necessary. It is important to use the correct path notation depending on the operating system. This would be `c:\\path\\to.exe` in Windows and `/path/to/executable` in Linux and macOS.
- **start_on_server** and **start_on_client –** Indicates whether an extension defined by the manifest should be run on the server or client or both.
- **virtual_channel_namespace –** This attribute specifies the namespace for virtual channels created by this extension. Although different vendors may create virtual channels with the same name, they should be in separate namespaces. Namespace 'dcv' is reserved, so it shouldn't be used.
- **userdata –** NICE DCV ignores this attribute, but the extension can access it via `GetManifestRequest`.

Extension executable

Extension manifest files define the `executable` file spawned by NICE DCV. NICE DCV will exchange messages with the extension using the `stdin` and `stdout` of the extension process.

- The NICE DCV client starts the extension once a NICE DCV connection is established. The executable runs in the context of the user who launched the client.
- The NICE DCV server starts the extension when a user logs in. The executable runs in the context of the logged-in user.

Digital signature

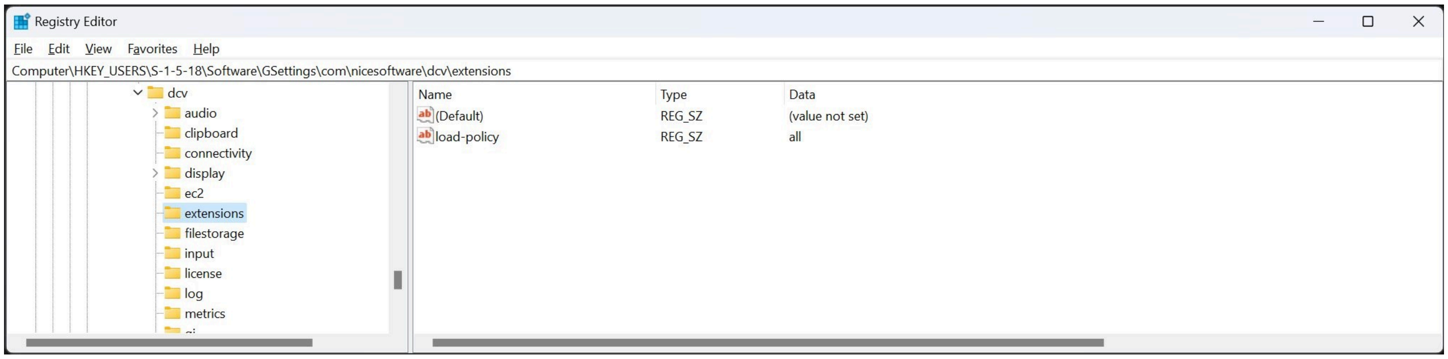
On Windows, NICE DCV starts only digitally signed extension executables. There is no digital signature verification on Linux and macOS. Digital signatures are verified using the `WinVerifyTrust` function with the `WINTRUST_ACTION_GENERIC_VERIFY_V2` parameter.

Important

During the development, it is possible to disable the verification for testing purposes. It is advisable not to set the following registry key outside of a development environment.

Adding the following registry key will allow DCV to load unsigned extensions executables on the server:

```
[HKEY_USERS\S-1-5-18\Software\GSettings\com\nicesoftware\dcv\extensions]
load-policy=all
```



Computer\HKEY_USERS\S-1-5-18\Software\GSettings\com\nicesoftware\dcv\extensions

Name	Type	Data
(Default)	REG_SZ	(value not set)
load-policy	REG_SZ	all

To start unsigned extensions on the client, use the following command line parameter:

```
--extensions-load-policy="all"
```

Installing and registering the extension

NICE DCV does not determine where extension executables should be located. However, to ensure that file system ACLs protect extensions from unauthorized modification, it is best to follow the guidelines for your specific operating system. On Windows, for instance, use the `Program Files` folder.

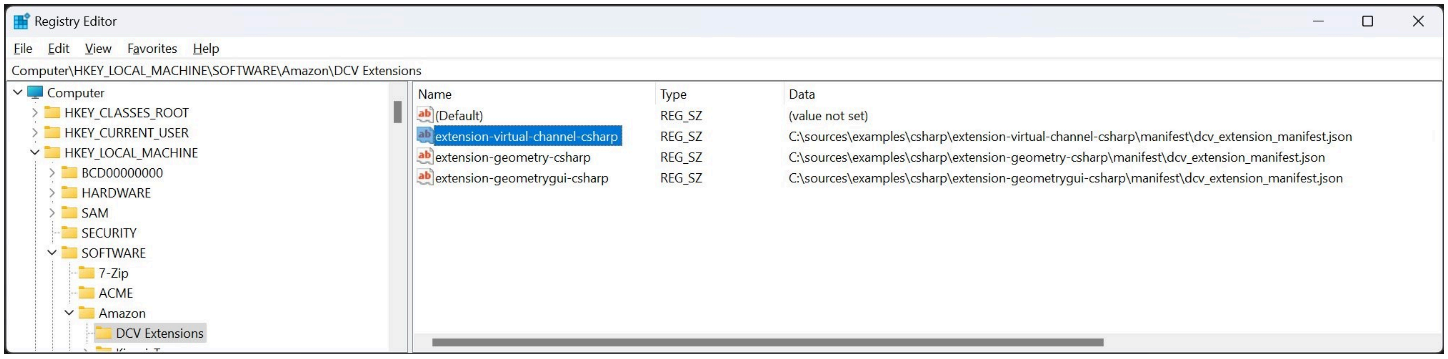
Registering the extension on Windows

On Windows, manifest files can be placed in any directory, including the same directory as extension executables. Add the string value to the registry key outlined below to register the extension. If the key does not exist, create it. If you remove the extension, do not delete the key, since this may break other third-party extensions. Value names must contain the name of the extension, and value data must contain the path to the manifest file.

On the client side, extensions could be registered per user or per machine. On the server side, extensions are registered only per-machine.

The per-machine key is `HKEY_LOCAL_MACHINE\SOFTWARE\Amazon\DCV Extensions`

The per-user key is `HKEY_CURRENT_USER\SOFTWARE\Amazon\DCV Extensions`



Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Amazon\DCV Extensions

- Computer
  - HKEY_CLASSES_ROOT
  - HKEY_CURRENT_USER
  - HKEY_LOCAL_MACHINE
    - BCD00000000
    - HARDWARE
    - SAM
    - SECURITY
    - SOFTWARE
      - 7-Zip
      - ACME
      - Amazon
        - DCV Extensions

Name	Type	Data
(Default)	REG_SZ	(value not set)
extension-virtual-channel-csharp	REG_SZ	C:\sources\examples\csharp\extension-virtual-channel-csharp\manifest\dcv_extension_manifest.json
extension-geometry-csharp	REG_SZ	C:\sources\examples\csharp\extension-geometry-csharp\manifest\dcv_extension_manifest.json
extension-geometrygui-csharp	REG_SZ	C:\sources\examples\csharp\extension-geometrygui-csharp\manifest\dcv_extension_manifest.json

Registering the extension on Linux

You can register the extension by placing the `.json` manifest in the folder outlined below. If the folder does not exist, create it. If you remove an extension, you should not remove the folder or any files in it, since that may break other extensions. The manifest file name must be unique to the extension.

On the client side, extensions could be registered per user or per machine. On the server side, extensions are registered only per-machine

The per-machine folder is: `/usr/share/dcvextensions/`

The per-user folder is: `~/.local/share/dcvextensions`

Registering the extension on macOS

You can register the extension by placing the `.json` manifest in the folder outlined below. If the folder does not exist, create it. If you remove an extension, you should not remove the folder or any files in it, since that may break other extensions. The manifest file name must be unique to the extension.

On the client side, extensions could be registered per user or per machine. On the server side, extensions are registered only per-machine

The per-machine folder is: `/Library/DCV Extensions/`

The per-user folder is: `Library/DCV Extensions/`

Permissions

NICE DCV has two new permissions to enable the execution of the extensions:

- `extensions-server`: Allows to start the installed extensions on the server machine.
- `extensions-client`: Allows to start the installed extensions on the client machine.

For more information about the permissions files, Refer to Working with permissions files in the NICE DCV Administrator Guide.

Protocol and framing

When NICE DCV components start the extension on either server or client side, DCV uses standard input (`stdin`) and standard output (`stdout`), also known as anonymous pipes, to communicate with the extension.

The NICE DCV component and extension are treated as peers and use the same wire protocol. Communication is bidirectional.

Extensions receive binary frames from NICE DCV from their standard input stream and must write binary frames for NICE DCV to their standard output stream.

Important

Extensions must refrain from writing any output to their standard error stream. Failure to do so may cause their process to be terminated.

Frame

A frame that is sent by either peer consists of a mandatory header followed by the payload message.

```
     FRAME
  +--------+------------------------------------------·--·--+
  | header | message                                        |
  +--------+------------------------------------------·--·--+
   4 bytes     variable length
```

Header

A binary header is consisting of a single 4 bytes little-endian unsigned integer, representing the size in bytes of the following message.

```
     HEADER
 +------------+------------+------------+------------+
 |byte 0 (LSB)|   byte 1   |   byte 2   |byte 3 (MSB)|
 +------------+------------+------------+------------+
```

Message

NICE DCV and NICE DCV Extension SDK use Protocol Buffers (protobuf) protocol to send messages in both directions. The definition of the messages are contained in the file `extensions.proto`.

Messages are exchanged in binary form. Also, the protobuf provides for serializers and deserializers to and from binary streams for many programming languages.

```
     MESSAGE
 +--------------------------------------------------·--·--+
 | bytes                                                  |
 +--------------------------------------------------·--·--+
   number of bytes as specified in the header
```

API Reference

All messages sent from the extensions to NICE DCV are of a type of request. Messages from NICE DCV to extensions can be of type response or event.

Requests from an extension will always receive a synchronous response from NICE DCV of either success in a single with a stated status value or failure with a stated reason.

Extensions may specify in each request that an optional `request_id` will be replicated in the corresponding response.

Some requests require asynchronous processing where a successful response means that the request is being processed and the final outcome will be delivered with an event message.

Event messages are also used to deliver notifications of conditions originating in the local NICE DCV client/server, the remote server/client, or the remote extension.

Example

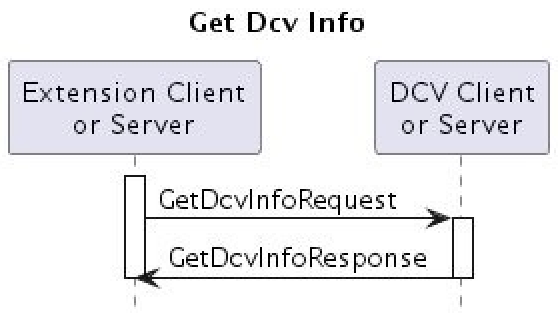
An example of an event message would be `StreamingViewsChangedEvent`

Topics

General API

GetDcvInfo

This API provides a generic information about the NICE DCV components, including the software version and architecture.



Helper structures:

```
VersionNumber
+ uint32 major
+ uint32 minor
+ uint32 revision

SoftwareInfo
+ string name
+ VersionNumber version
+ string os
+ string arch
+ string hostname
```

Request message:

```
GetDcvInfoRequest
(no fields)
```

Response message:

```
GetDcvInfoResponse
+ DcvRole dcv_role
+ int64 dcv_process_id
+ SoftwareInfo server_info
+ SoftwareInfo client_info
```

- **dcv_role**: Enumeration, either Server or Client, tells if the parent process is DCV Client or a DCV Server.
- **dcv_process_id**: The pid of the parent process: the DCV viewer on clients, the DCV user agent process on servers.
- **server_info**: Information on the DCV Server, either on the local side or remote side of the DCV connection.
- **client_info**: Information on the DCV Client, either on the local side or remote side of the DCV connection.

GetManifest

This API describes the location of the manifest file that was used to register this specific extension.

Request message:

```
GetManifestRequest
(no fields)
```

Response message:

```
GetManifestResponse
+ string manifest_path
```

- **manifest_path**: Path to the manifest file used to register the extension.

Virtual Channel API

SetupVirtualChannel

When a client extension and a server extension want to exchange data, they can setup a virtual channel between each other.

Both parties will issue a `SetupVirtualChannelRequest` message and both will wait for a `VirtualChannelReadyEvent` message. The synchronous response contains the name of the relay (named pipe on Windows, abstract UNIX domain socket on Linux and UNIX domain socket on macOS) over which the extensions will exchange data with NICE DCV and an authentication token used by NICE DCV to validate the process that is connecting to the relay endpoint.

- The caller must establish the communication with DCV.

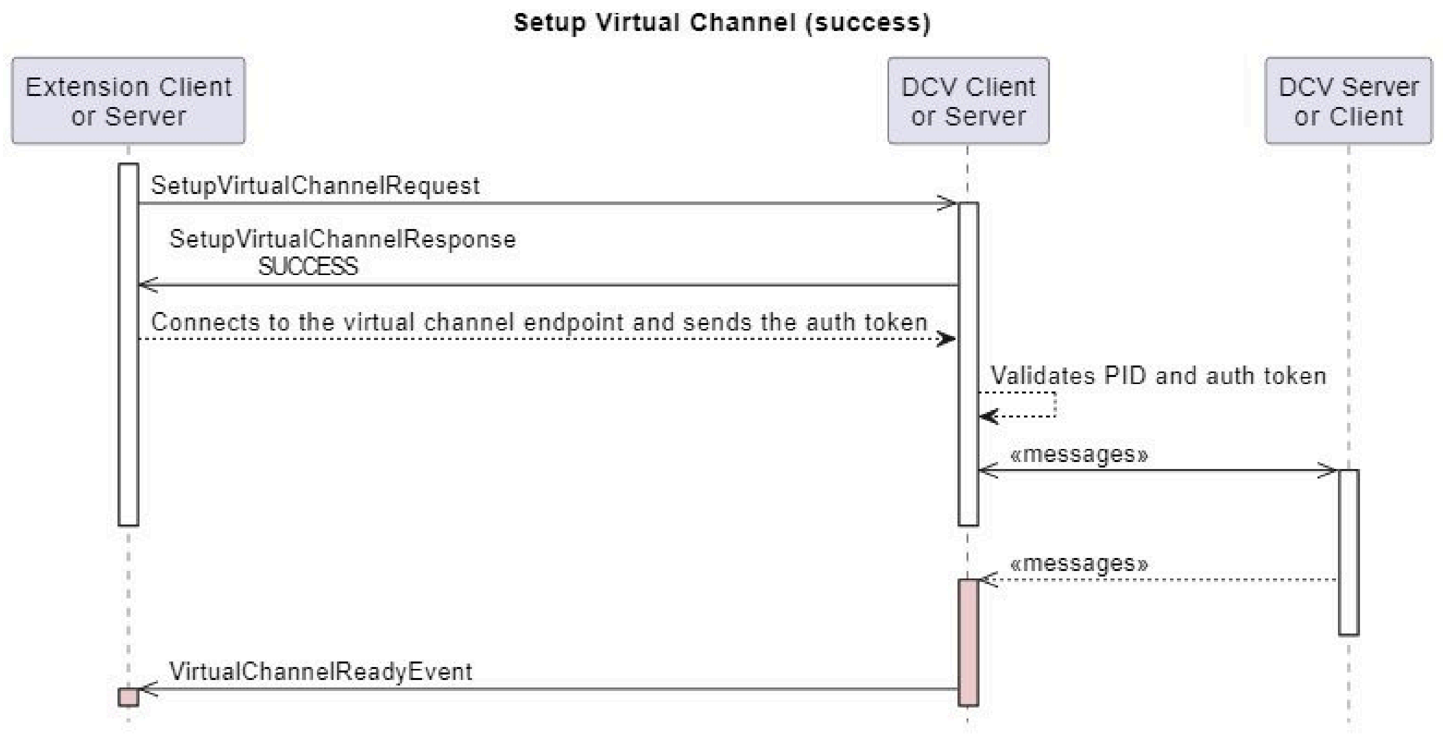
  For Windows, open the client side of the named pipe.

  For Linux, open the abstract UNIX domain socket.

  For macOS, open the UNIX domain socket.
- Send the authentication token to the data channel.
- Wait for the asynchronous event on the control channel before writing any other data to it.

The caller must specify the ID of the process that will open the client side of the relay, NICE DCV will reject any attempt to connect to the relay from another process. Should the extension use the virtual channel from another process (e.g. a daemon process crashes and it is re-spawned), it will need to undergo a new virtual channel setup procedure.

The maximum number of open virtual channels (established or pending) for one extension is four, further setup requests will fail.


Setup Virtual Channel (success)
Extension Client or Server
DCV Client or Server
DCV Server or Client
SetupVirtualChannelRequest
SetupVirtualChannelResponse SUCCESS
Connects to the virtual channel endpoint and sends the auth token
Validates PID and auth token
«messages»
«messages»
VirtualChannelReadyEvent

Request message:

```
SetupVirtualChannelRequest
+ string virtual_channel_name
+ int64 relay_client_process_id
```

- **virtual_channel_name**: Specific name of the virtual channel, the namespace is taken from the manifest.
- **relay_client_process_id**: Pid of the process that will open the client side of the relay.

Response message:

```
SetupVirtualChannelResponse
+ string virtual_channel_name
+ string relay_path
+ int64 relay_client_process_id
+ bytes virtual_channel_auth_token
```

- **virtual_channel_name**: Specific name of the virtual channel to be set up. The namespace is taken from the manifest.
- **relay_path**: Full path of the relay to the virtual channel. On Windows, it is the name of a named pipe. On Linux, it is the name of the abstract UNIX domain socket. On macOS, it is the name of the UNIX domain socket.

  Example

  For example:`\\.\pipe\dcvextensions_random`
- **relay_client_process_id**: The pid of the process that runs the server side of the relay client: the NICE DCV viewer on clients, the NICE DCV server process on servers.
- **virtual_channel_auth_token**: An authentication token the process has to send on the relay channel. DCV will use it to validate the connecting process.

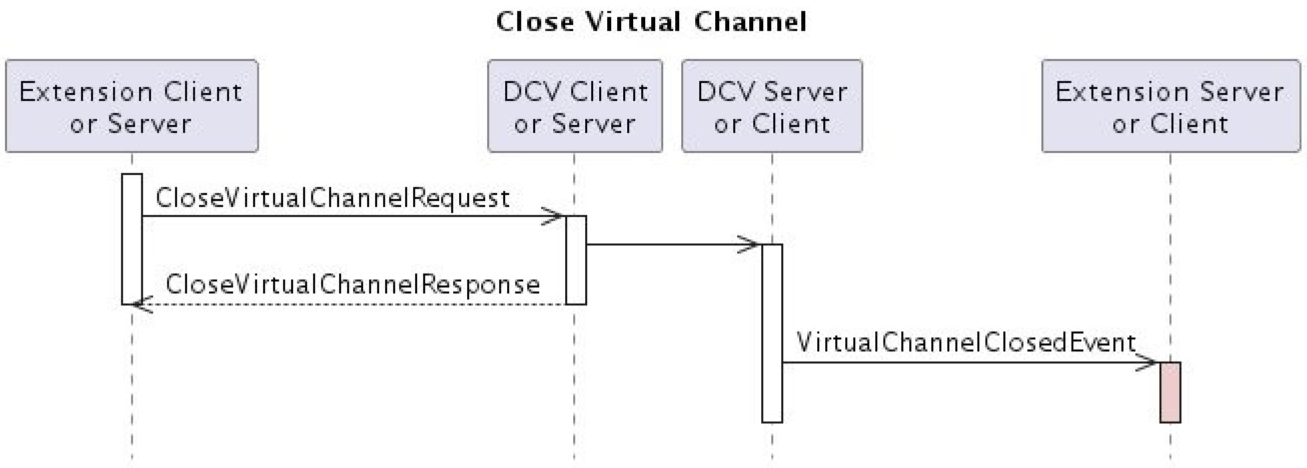
Event message:

```
VirtualChannelReadyEvent
+ string virtual_channel_name
```

- **virtual_channel_name**: Specific name of the virtual channel, the namespace is taken from the manifest.

CloseVirtualChannel

One party (either the client extension or the server extension) can initiate the closure of a virtual channel with a `CloseVirtualChannelRequest`, the other party will be notified with a `VirtualChannelClosedEvent`.


Close Virtual Channel
Extension Client or Server
DCV Client or Server
DCV Server or Client
Extension Server or Client
CloseVirtualChannelRequest
CloseVirtualChannelResponse
VirtualChannelClosedEvent

Request message:

```
CloseVirtualChannelRequest
+ string virtual_channel_name
```

- **Request message:virtual_channel_name**: Specific name of the virtual channel, the namespace is taken from the manifest.

Response message:

```
CloseVirtualChannelResponse
+ string virtual_channel_name
```

- **virtual_channel_name**: Specific name of the virtual channel to be set up, the namespace is taken from the manifest.

Event message:

```
VirtualChannelClosedEvent
+ string virtual_channel_name
```

- **virtual_channel_name**: Specific name of the virtual channel, the namespace is taken from the manifest.

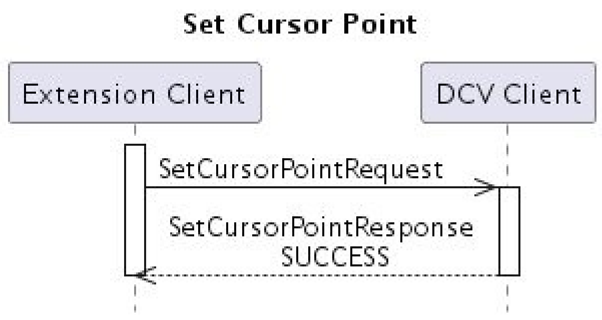
Geometry API

SetCursorPoint

Note

This API is supported by, and only available on, NICE DCV Windows clients only.

Move the local cursor pointer to a specific position over a streaming area of the NICE DCV client. Extensions issue a `SetCursorPointRequest` and receive a synchronous `SetCursorPointResponse`.



Helper structures:

```
Point
+ int32 x
+ int32 y
```

Request message:

```
SetCursorPointRequest
+ Point point
```

- **point**: New position for the cursor pointer expressed in local virtual screen coordinates.

  Example

  On Windows the coordinates in the virtual screen are expressed in physical pixels (no custom DPI applied) with the origin corresponding to the top left corner of the main display, displays to the left or to the top have negative coordinates.

Response message:

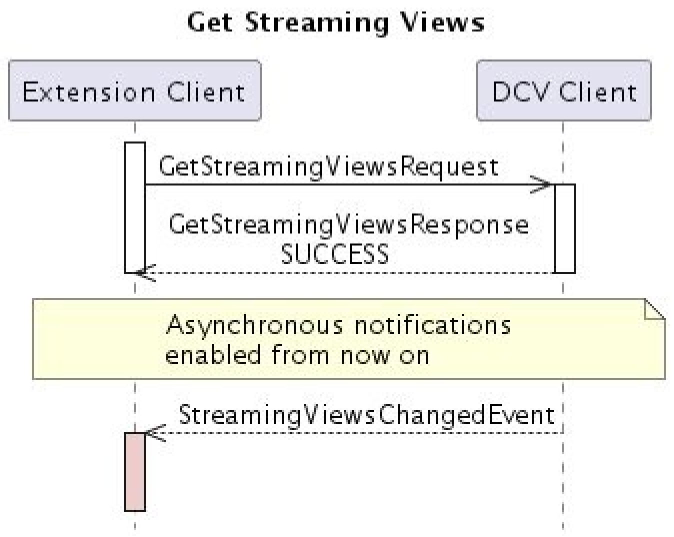
```
SetCursorPointResponse
(no fields)
```

GetStreamingViews

This API is available only on DCV client.

Extensions issue a `GetStreamingViewRequest` and receive a synchronous `GetStreamingViewsResponse`.

After the first GetStreamingViewsRequest is issued DCV will also start sending asynchronous `StreamingViewsChangedEvent` messages whenever the geometry of the local streaming views is changed (e.g. the client window is resized or moved) or the geometry of the remote desktop is changed (e.g. a monitor is added or removed).


Get Streaming Views
Extension Client
DCV Client
GetStreamingViewsRequest
GetStreamingViewsResponse
SUCCESS
Asynchronous notifications enabled from now on
StreamingViewsChangedEvent

Helper structures:

```
Point
+ int32 x
+ int32 y
```

```
Rect
+ int32 x
+ int32 y
+ uint32 width
+ uint32 height
```

```
StreamingView
+ int32 view_id
+ Rect local_area
+ double zoom_factor
+ Point remote_offset
+ bool has_focus
```

```
StreamingViews
+ StreamingView[] streaming_view
+ bool has_focus
```

- **view_id**: The view ID, as returned in `IsPointInsideStreamingViewsResponse`
- **local_area**: Coordinates of the local streaming view in the local virtual screen.

  Example

  On Windows the coordinates in the virtual screen are expressed in physical pixels (no custom DPI applied) with the origin corresponding to the top left corner of the main display, displays to the left or to the top have negative coordinates.
- **zoom_factor**: Local over remote, e.g. 2.0 means that local view is twice as big as remote area
- **remote_offset**: Remote virtual screen coordinates, regardless of the remote operating system the origin is on the top left corner of the smallest rectangle containing all remote displays so remote virtual screen coordinates are always positive.
- **has_focus**: StreamingView.has_focus indicate whether an individual streaming view has has the keyboard focus or not.

  Example

  StreamingViews.has_focus indicate whether the client window has has the keyboard focus or not.

Request message:

```
GetStreamingViewsRequest
(no fields)
```

Response message:

```
GetStreamingViewsResponse
+ StreamingViews streaming_views
```

Event message:

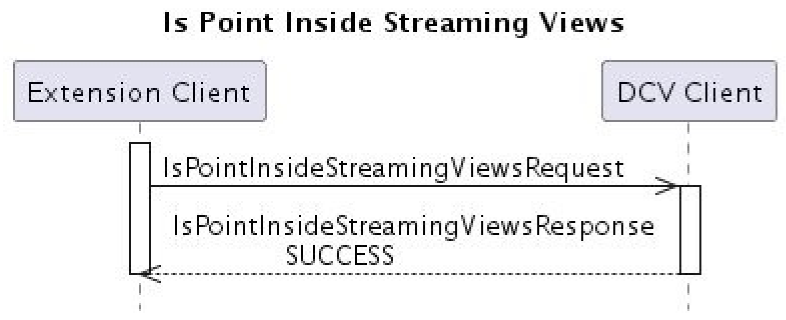
```
StreamingViewsChangedEvent
+ StreamingViews streaming_views
```

IsPointInsideStreamingViews

This API is available only on NICE DCV client.

Query whether a position on the local desktop of the client host lies inside a visible part of a streaming area of the NICE DCV client, i.e. it shows a pixel of the remote desktop. In case a point lies in a streaming view, the id of the streaming view is returned (to be matched to id's returned in `GetStreamingViewsResponse` or `GetStreamingViewsChangedEvent`), in case the point lies outside all streaming views -1 is returned.

Extensions issue a `IsPointInsideStreamingViewsRequest` and receive a synchronous `IsPointInsideStreamingViewsResponse`.


Is Point Inside Streaming Views
Extension Client
DCV Client
IsPointInsideStreamingViewsRequest
IsPointInsideStreamingViewsResponse
SUCCESS

Helper structures:

```
Point
+ int32 x
+ int32 y
```

Request message:

```
IsPointInsideStreamingViewsRequest
+ Point point
```

- **point**: Expressed in local virtual screen coordinates.

  Example

  On Windows the coordinates in the virtual screen are expressed in physical pixels (no custom DPI applied) with the origin corresponding to the top left corner of the main display, displays to the left or to the top have negative coordinates.

Response message:

```
IsPointInsideStreamingViewsRequest
+ int32 view_id
```

- **view_id**: Identification of the streaming view in which the point lies (from `GetStreamingViewsResponse` or `GetStreamingViewsChangedEvent`), or -1 if the point lies outside all local views.

Release Notes and Document History for NICE DCV Extension SDK

This page provides the release notes and document history for NICE DCV Extension SDK.

Topics

NICE DCV Extension SDK Release Notes

This section provides release notes for the NICE DCV Extension SDK by release date.

Topics

1.0.0 — April 3, 2023

Version	Release notes
• Semantic version: 1.0.0	Initial release of the NICE DCV Extension SDK

Document History

The following table describes the documentation for this release of NICE DCV Extension SDK.

Change	Description	Date
Initial release	First publication of this content.	April 3, 2023