Nokia Service Router Linux

Release 24.7

Event Handler Guide

3HE 20667 AAAA TQZZA
Edition: 01
July 2024

Nokia is committed to diversity and inclusion. We are continuously reviewing our customer documentation and consulting with standards bodies to ensure that terminology is inclusive and aligned with the industry. Our future customer documentation will be updated accordingly.

Table of contents

1 About this guide

This document describes the functionality and configuration for the event handler framework on the Nokia Service Router Linux (SR Linux). A configuration example is provided showing how the event handler framework can be used to support operational groups.

This document is intended for marketing personnel, network technicians, administrators, operators, service providers, and others who need a basic understanding of how event handler works.

Note:
This manual covers the current release and may also contain some content that will be released in later maintenance loads. See the *SR Linux Release Notes* for information about features supported in each load.

Configuration and command outputs shown in this guide are examples only; actual displays may differ depending on supported functionality and user configuration.

1.1 Precautionary and information messages

The following are information symbols used in the documentation.

DANGER: Danger warns that the described activity or situation may result in serious personal injury or death. An electric shock hazard could exist. Before you begin work on this equipment, be aware of hazards involving electrical circuitry, be familiar with networking environments, and implement accident prevention procedures.

WARNING: Warning indicates that the described activity or situation may, or will, cause equipment damage, serious performance problems, or loss of data.

Caution: Caution indicates that the described activity or situation may reduce your component or system performance.

Note: Note provides additional operational information.

Tip: Tip provides suggestions for use or best practices.

1.2 Conventions

SR Linux documentation uses the following command conventions.

- **Bold** type indicates a command that the user must enter.
- Input and output examples are displayed in `Courier` text.
- An open right-angle bracket indicates a progression of menu choices or simple command sequence (often selected from a user interface). Example: **start** > **connect to**.

- A vertical bar (|) indicates a mutually exclusive argument.
- Square brackets ([ ]) indicate optional elements.
- Braces ({ }) indicate a required choice. When braces are contained within square brackets, they indicate a required choice within an optional element.
- *Italic* type indicates a variable.

Generic IP addresses are used in examples. Replace these with the appropriate IP addresses used in the system.

2 What's new

There have been no updates in this document since it was last released.

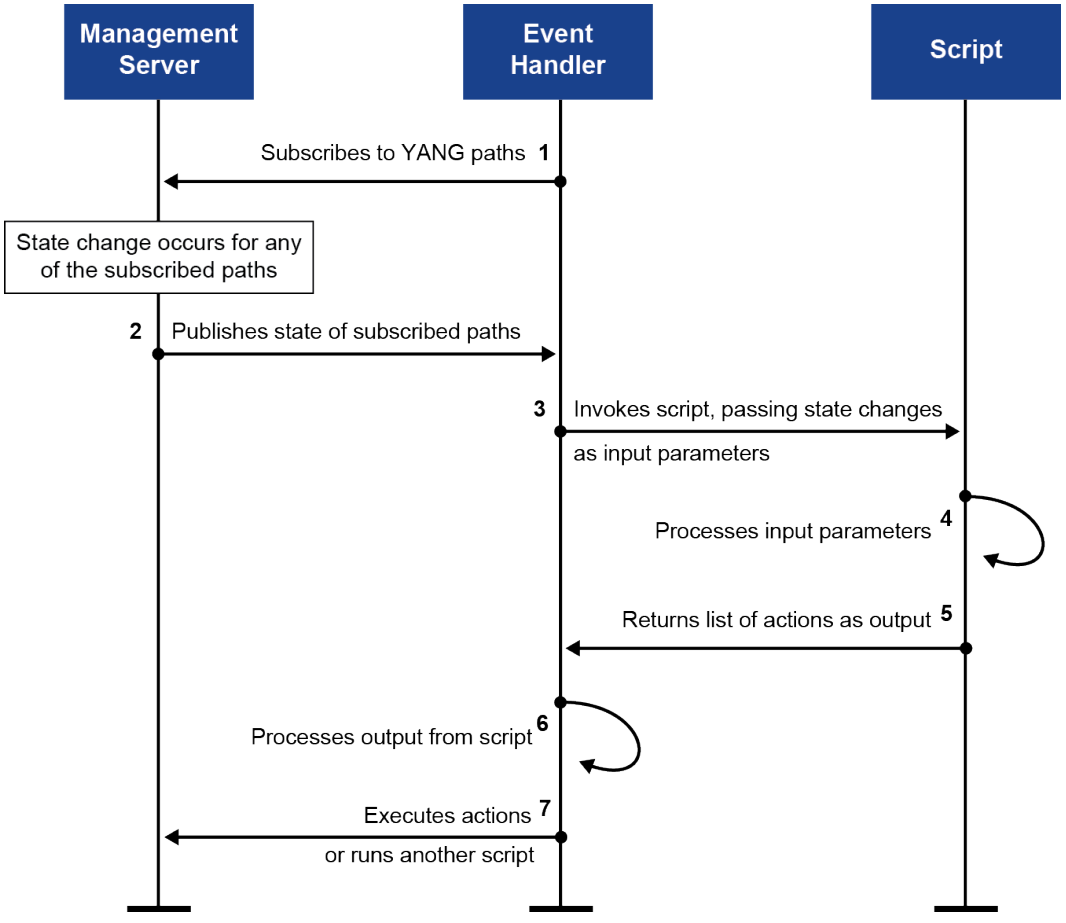
3 Event handler overview

Event handler is a framework that enables SR Linux to react to specific system events, using programmable logic to define the actions taken in response to the events.

The event handler framework allows you to write custom scripts that are invoked when specific events occur, such as when a port goes operationally down. The scripts can generate a list of actions for the SR Linux device to execute. The actions can include updating the SR Linux configuration, changing the operational state of a group of ports, executing a **tools** command, or running another script.

SR Linux supports event handler via the `event_mgr` process, which operates as the `srlinux` user. The following diagram shows the interaction between the SR Linux management server, event handler (via the `event_mgr` process), and the script.

Figure 1: Event handler framework



The event handler framework operates in the following sequence:

1. Event handler is configured to subscribe to a set of YANG paths, for example the operational state of a set of interfaces.

2. The SR Linux management server publishes the state of the paths to which event handler is subscribed.
3. In response to the state change, event handler invokes a MicroPython script, supplying path information and configured options for the script to use as input.
4. The script processes the information supplied by event handler.
5. The script returns a list of actions that is passed to event handler. Possible actions include setting an interface operationally down, running a **tools** command, or executing another script.
6. Event handler processes the list of actions returned by the script.
7. Event handler executes the actions on the SR Linux device, or runs another script if an action specifies to do so.

To configure event handler, you configure an event handler instance, which specifies the paths to monitor, the name of the script to run, and options to supply to the script. See Event handler configuration.

Event handler supplies input to a script as a JSON string. See Event handler scripts for details about how the script input is formatted, how it is processed by a script, and the actions that a script can return to event handler.

One possible use for event handler is the ability to change the operational state of one group of ports based on the state of another group of ports. This type of usage is known as operational groups. Operational groups overview describes this use case. Configuring event handler for operational groups provides an example configuration.

4 Event handler configuration

To configure an event handler instance, you specify the paths to monitor, the name of the MicroPython script to invoke, and object-value pairs to be used as options in the script. When a state change occurs for the monitored paths, event handler invokes the script, passing the state information and options in a JSON string as input to the script.

Example

The following is an example configuration for an event handler instance:

```
--{ candidate shared default }--[  ]--
# info system event-handler instance opergrp
    system {
        event-handler {
            instance opergrp {
                admin-state enable
                upython-script oper-group.py
                paths [
                    "interface ethernet-1/55 oper-state"
                    "interface ethernet-1/56 oper-state"
                ]
                options {
                    object down-links {
                        values [
                            ethernet-1/1
                        ]
                    }
                    object required-num-up-links {
                        value 2
                    }
                }
            }
        }
    }
```

In this example, the event handler monitors the operational state of the uplinks defined in the `paths` statement, `ethernet-1/55` and `ethernet-1/56`. If the `oper-state` for either uplink changes to `down`, then the `oper-state` of the downstream link in the `down-links` options list, `ethernet-1/1`, is set to `down`.

To do this, event handler invokes the MicroPython script defined in the `upython-script` statement, `oper-group.py`. Event handler supplies the state of the uplinks, the number of uplinks required to be up, and the associated downlinks as input to the script; this input is supplied in the form of a JSON string; for example:

```
{
    "paths": [
        {
            "path": "interface ethernet-1/55 oper-state",
            "value": "down"
        },
        {
            "path": "interface ethernet-1/56 oper-state",
            "value": "down"
```

```
        }
    ],
    "options": {
        "required-num-up-links": "2",
        "down-links": [
            "ethernet-1/1",
        ]
    }
}
```

The logic for counting the number of operationally up uplinks and setting downlinks to operationally down is kept within the `oper-group.py` script. After processing the input, the script returns a list of actions to event handler for execution on the SR Linux device. See Event handler scripts for more information.

Specifying paths to monitor

The `paths` statement specifies the objects for the event handler instance to monitor. Paths are configured in CLI notation and specify any leaf or leaf-list available in the state datastore. You can use ranges and wildcards in the path.

The following examples show possible `paths` configurations in an event handler instance:

- `"interface ethernet-1/1 oper-state"`
  Monitors changes in the `oper-state` for interface `ethernet-1/1`
- `"interface ethernet-1/{1..12} oper-state"`
  Monitors changes in the `oper-state` for interface `ethernet-1/1` through `ethernet-1/12`
- `"interface ethernet-1/* oper-state"`
  Monitors changes in the `oper-state` for all interfaces on a line card
- `"interface * oper-state"`
  Monitors changes in the `oper-state` for all interfaces on the SR Linux device

Specifying options

The `options` statement contains a set of user-defined objects and associates either a single `value` or list of `values` to them. The object-value pairs are carried as input to the MicroPython script, allowing you to potentially pass configuration. Event handler passes the `options` to the script as JSON strings; the objects do not need to follow any particular schema.

The following example defines object-value pairs in the `options` statement of an event handler instance:

```
--{ candidate shared default }--[  ]--
# info system event-handler instance opergrp options
    system {
        event-handler {
            instance opergrp {
                options {
                    object down-links {
                        values [
                            ethernet-1/1
                            ethernet-1/2
                        ]
                    }
                    object required-num-up-links {
                        value 1
                    }
                }
```

```
            }
        }
    }
```

In the example, the interfaces in the `down-links` object are specified as a list of `values`, and the number in the `required-num-up-links` object is specified as a single `value`.

Specifying a MicroPython script

To specify the MicroPython script, use the `upython-script` statement. This statement refers to a MicroPython script in one of the following locations:

- `/etc/opt/srlinux/eventmgr` for user-provided scripts
- `/opt/srlinux/eventmgr` for Nokia-provided scripts

No other directories can be used for event handler scripts. See Event handler scripts for information about how event handler supplies input to a script and how the script processes the input and returns a list of actions to the event handler for execution.

4.1 Configuring the user for event handler operations

Procedure

By default, event handler configuration operations executed via the `set-cfg-path` and `delete-cfg-path` actions, scripts executed via the `run-script` action, and **tools** commands executed via the `set-tools-path` action are all performed by the SR Linux `admin` user. You can change the user that performs these operations.

The user must be a local user; that is, a user configured within the SR Linux CLI. Linux users and remote users cannot be configured to execute event handler operations. See "User types" in the *SR Linux Configuration Basics Guide* for information about each user type.

The user's role must have the appropriate privileges to perform the `set-cfg-path`, `delete-cfg-path`, and `set-tools-path` actions required for the event handler operations. See "Authorization using role-based access control" in the *SR Linux Configuration Basics Guide* for information about configuring roles.

The configured user executes operations for all event handler instances. If a user is not configured to execute event handler operations, they are executed by local SR Linux user `admin`.

Example: Configure an SR Linux local user to execute event handler operations

The following example creates a local SR Linux user and configures the user to execute operations for event handler instances.

```
--{ * candidate shared default }--[ ]--
# system aaa authentication user EHexec password sr1l234
```

```
--{ * candidate shared default }--[ ]--
# info system event-handler
    system {
        event-handler {
            run-as-user EHexec
        }
    }
```

Example: Configure the user's role to allow execution of event handler operations

The following example configures a role that gives write privileges to all paths (/) and assigns that role to the user configured to execute event handler operations.

```
--{ candidate shared default }--[  ]--
# info system aaa authorization
    system {
        aaa {
            authorization {
                role EHexecRole {
                }
            }
        }
    }
```

```
--{ * candidate shared default }--[  ]--
# info system configuration role EHexecRole
    system {
        configuration {
            role EHexecRole {
                rule / {
                    action write
                }
            }
        }
    }
```

```
--{ * candidate shared default }--[  ]--
# info system aaa authentication
    system {
        aaa {
            authentication {
                authentication-method [
                    local
                ]
                user EHexec {
                    password $ar2$kYDzvtHbUuc=$VINLYMbZ8f8u4gMtuPnSVQ==
                    role [
                        EHexecRole
                    ]
                }
            }
        }
    }
```

4.2 Displaying event handler information

Procedure

Use the **info from state** command to display information about an event handler instance, including any errors.

Example

```
--{ running }--[  ]--
# info from state system event-handler instance opergrp
```

```
    system {
        event-handler {
            instance opergrp {
                admin-state enable
                upython-script oper-group.py
                oper-state up
                paths [
                    "interface ethernet-1/1 oper-state"
                    "interface ethernet-1/4 oper-state"
                ]
                options {
                    object down-links {
                        values [
                            ethernet-1/3
                            ethernet-1/8
                        ]
                    }
                    object required-num-up-links {
                        value 2
                    }
                }
                last-execution {
                    start-time now
                    end-time now
                    upython-duration 1
                    input "{\"paths\":[{\"path\":\"interface ethernet-1/1 oper-state\",
\"value\":\"up\"},{\"path\":\"interface ethernet-1/4 oper-state\",\"value\":\"up\"}],\
"options\":{\"down-links\":[\"ethernet-1/3\",\"ethernet-1/8\"],\"required-num-up-links\":\
"2\"},\"persistent-data\":{\"last-state\":\"up\"}}"
                    output "{\"actions\": [{\"set-ephemeral-path\": {\"path\": \"interface
 ethernet-1/3 oper-state\", \"value\": \"up\"}}, {\"set-ephemeral-path\": {\"path\": \
"interface ethernet-1/8 oper-state\", \"value\": \"up\"}}], \"persistent-data\": {\"last-
state\": \"up\"}}"
                    stdout-stderr ""
                }
                last-errored-execution {
                    oper-down-reason admin-disabled
                    oper-down-reason-detail ""
                    start-time "26 seconds ago"
                    end-time "25 seconds ago"
                    upython-duration 0
                    input "{\"paths\":[{\"path\":\"interface ethernet-1/1 oper-state\",\
"value\":\"up\"},{\"path\":\"interface ethernet-1/4 oper-state\",\"value\":\"down\"}],\
"options\":{\"down-links\":[\"ethernet-1/3\",\"ethernet-1/8\"],\"required-num-up-links\":\
"2\"},\"persistent-data\":{\"last-state\":\"down\"}}"
                    output "{\"actions\": [{\"set-ephemeral-path\": {\"path\": \"interface
 ethernet-1/3 oper-state\", \"value\": \"up\"}}, {\"set-ephemeral-path\": {\"path\": \
"interface ethernet-1/8 oper-state\", \"value\": \"up\"}}], \"persistent-data\": {\"last-
state\": \"up\"}}"
                    stdout-stderr ""
                }
                statistics {
                    upython-duration 516
                    execution-count 1643
                    execution-successes 1642
                    execution-errors 1
                }
            }
        }
    }
```

Example

You can clear the statistics for the event handler instance using a **tools** command. For example:

```
--{ running }--[  ]--
# tools system event-handler instance opergrp statistics clear
```

4.3 Error handling for event handler

Whenever calling the `event_handler_main()` function for a script results in an error, the event handler instance `oper-state` is set to `down`, and the `oper-down-reason` under `last-errored-execution` indicates the reason for the failure. The `oper-down-reason-detail` is populated from `stderr` and displays any additional details. The relevant error counters `execution-timeouts` and `execution-errors` are incremented.

Whenever a script exceeds the 10-second execution timer (the amount of time it takes for a call to `event_handler_main()` to return a list of actions, and for `event_mgr` to execute the actions) the function call is terminated, and the `execution-timeouts` error counter is incremented.

Any time an instance experiences a failure, any pre-existing `set-ephemeral-path` actions are cleared, resulting in the modified objects reverting back to their original state.

If a script fails, it is automatically retried after a 10-second delay. During the 10-second delay, the `oper-reason` reflects the most recent `oper-state down` transition. After the 10-second delay, the instance moves to `oper-state starting`, before moving to `oper-state down` or `oper-state up` depending on the execution of the `event_handler_main()` function.

The 10-second delay and retry continues until the script execution is successful. For the following `oper-reason` errors, the instance stays at `oper-state down` until a user intervenes, with no retry:

- `failed-to-compile`
- `script-unavailable`
- `system-error`
- `missing-function`

5 Event handler scripts

An event handler script contains the logic that operates on the input JSON string passed to it by the `event_mgr` process each time a state change is detected for the paths monitored by the event handler instance.

Event handler scripts are executed by a MicroPython interpreter, and therefore have a limited set of modules available in the standard libraries compared to Python. You can use a regular Python interpreter to write the scripts for the event handler as long as you use standard libraries available for MicroPython.

Script input

Whenever a state change is detected for any of the monitored paths defined in the event handler instance, event handler calls the MicroPython script referenced in the event handler instance configuration. Event handler calls the `event_handler_main()` function in the script, passing it a JSON string indicating the current values of the monitored paths, as well as any other configured options.

Using the example in Event handler configuration, when the `oper-state` for the two links defined in `paths` changes to `down`, the `event_mgr` process generates the following JSON string and passes it to a script (named `oper-group.py` in the example) as input:

```
{
    "paths": [
        {
            "path": "interface ethernet-1/55 oper-state",
            "value": "down"
        },
        {
            "path": "interface ethernet-1/56 oper-state",
            "value": "down"
        }
    ],
    "options": {
        "required-num-up-links": "1",
        "down-links": [
            "ethernet-1/1",
            "ethernet-1/2",
        ]
    }
}
```

Script processing

The JSON string generated by the event handler is processed by a MicroPython script. In the following example, the script takes the state of the links in the supplied `paths` objects, and based on the `required-num-up-links` option:value pair, decides whether the links in the `down-links` option:value pair should be up or down. The script then generates a response consisting of actions for the SR Linux device to execute.

```
import sys
import json

# count_up_uplinks returns the number of monitored uplinks that have oper-state=up
def count_up_uplinks(paths):
```

```
    up_cnt = 0
    for path in paths:
        if path.get("value", "down") == "up":
            up_cnt = up_cnt + 1
    return up_cnt

# required_up_uplinks returns the value of the `required-up-uplinks` option
def required_up_uplinks(options):
    return int(options.get("required-up-uplinks", 1))

# main entry function for event handler
def event_handler_main(in_json_str):
    # parse input json string passed by event handler
    in_json = json.loads(in_json_str)
    paths = in_json["paths"]
    options = in_json["options"]
    num_up_uplinks = count_up_uplinks(paths)
    downlinks_new_state = (
        "down" if num_up_uplinks < required_up_uplinks(options) else "up"
    )

    # add `debug="true"` option to event-handler configuration to output parsed parameters
    if options.get("debug") == "true":
        print(
            f"num of required up uplinks = {required_up_uplinks(options)}\n\
detected num of up uplinks = {num_up_uplinks}\n\
downlinks new state = {downlinks_new_state}"
        )

    response_actions = []

    for downlink in options.get("down-links", []):
        response_actions.append(
            {
                "set-ephemeral-path": {
                    "path": "interface {downlink} oper-state",
                    "value": downlinks_new_state,
                }
            }
        )

    response = {"actions": response_actions}
    return json.dumps(response)
```

This script is an example of using event handler with the SR Linux operational groups feature. See Script processing for details about how each function of this script is processed.

Persistent-data object

The `persistent-data` object allows a script to maintain state between executions; for example, to count the number of times a particular action has been taken. The `persistent-data` object may be included in the JSON string returned by the script when invoked by event handler.

For example, if a script needs to count the number of times it has taken an action, it can return the following counter:

```
{
-- snip --
    "persistent-data": {
        "action-count": 5
    }
}
```

The next time event handler invokes the script, it includes the following as part of the input supplied to the script:

```
{
-- snip --
    "persistent-data": {
        "action-count": 5
    }
}
```

If configured to do so, the script can read this input and increment the counter, potentially returning the following:

```
{
-- snip --
    "persistent-data": {
        "action-count": 6
    }
}
```

Script output

The MicroPython script returns a single parameter, which is a JSON string with a structure expected by event handler. The script output contains a list of actions and `persistent-data` objects (if configured), which are passed to `event_mgr` for processing. See Actions for descriptions of the supported actions.

Using the example from Event handler configuration, if the script determines that the link in the `down-links` option must be brought down, it sends the following output JSON string to the Event Handler:

```
{
    "actions": [
        {
            "set-ephemeral-path": {
                "path": "/interface ethernet-1/1 oper-state",
                "value": "down",
            }
        }
    ]
}
```

Reloading the script

Whenever the configuration for an Event Handler instance changes, or the Event Handler instance is administratively disabled and re-enabled, any running calls to the previous main function are terminated, and the script is reloaded, losing any state information in the process. You can also reload the script manually using a **tools** command.

For example, the following command reloads the `oper-group.py` script:

```
--{ running }--[  ]--
# tools system event-handler instance oper-group.py reload
```

5.1 Actions

You can specify the following actions in a MicroPython script used with event handler:

- `set-ephemeral-path`
  Make an ephemeral change to a state leaf
- `set-cfg-path`
  Make a persistent change to the SR Linux configuration
- `delete-cfg-path`
  Delete a path in the SR Linux configuration
- `set-tools-path`
  Execute an SR Linux **tools** command
- `run-script`
  Issue a command to execute another script
- `reinvoke-with-delay`
  Re-invoke the script following a specified delay time
- `always-execute`
  Always process an action, regardless of whether the same action was processed previously

The `event_mgr` process expects the structure of the output JSON string from the script to adhere to the following schema:

```
{
    "actions": [
        {
            "set-ephemeral-path": {
                "path": "",
                "value": "",
                "always-execute": false
            }
        },
        {
            "set-cfg-path": {
                "path": "",
                "json-value": {},
                "always-execute": false
            }
        },
        {
            "delete-cfg-path": {
                "path": "",
                "always-execute": false
            }
        },
        {
            "set-tools-path": {
                "path": "",
                "json-value": {},
                "always-execute": false
            }
        },
        {
            "run-script": {
                "cmdline": "",
                "always-execute": false
            }
        },
        {
            "reinvoke-with-delay": 5000
        }
    ],
```

```
    "persistent-data": {
        "last-state-up": false
    }
}
```

Actions are processed in the order they are defined above, in that `set-ephemeral-path` is processed first, and `reinvoke-with-delay` last. The only exceptions are for `set-cfg-path` and `delete-cfg-path`, which are processed in the order they are received.

set-ephemeral-path

Makes an ephemeral change to a state leaf.

This action can be used to ephemerally change the `oper-state` for a set of interfaces based on some criteria (for example, the number of uplinks in `up` state).

- `path` is a leaf list in CLI notation that refers to the `oper-state` setting for a set of interfaces. Wildcards are not supported; ranges are supported.
- `value` can be either `down` or `up`.

The following example changes the `oper-state` for a set of four interfaces to `down`:

```
{
    "actions": [
        {
            "set-ephemeral-path": {
                "path": "interface ethernet-1/{1..4} oper-state",
                "value": "down"
            }
        }
    ]
}
```

If more than one event handler instance sets `oper-state` for the same interface, actions changing the `oper-state` to `down` override actions changing the `oper-state` to `up`.

Note:
In the current release, only the interface `oper-state` is valid as a path used with `set-ephemeral-path`.

set-cfg-path

Makes a persistent change to the SR Linux configuration.

The change is applied to the running configuration, but it is not saved to the startup configuration unless an explicit **save startup** command is executed.

- `path` is a leaf or leaf-list in CLI notation. Wildcards are not supported; ranges are supported
- `value` is a configuration setting relevant to the `path`.

The following example uses the `set-cfg-path` action to configure the `admin-state` for an interface. The `path` contains the CLI command to set the `admin-state`, and the `value` contains the setting to apply to the interface.

```
{
    "actions": [
        {
            "set-cfg-path": {
```

```
                "path": "interface ethernet-1/3 admin-state",
                "value": "disable"
            }
        },
    ]
}
```

delete-cfg-path

Deletes a path in the SR Linux configuration.

The change is applied to the running configuration, but it is not saved to the startup configuration unless an explicit **save startup** command is executed.

- `path` is a leaf or leaf-list in full, expanded CLI notation. No wildcards or ranges are supported. Deleted settings are reset to their default values where applicable.

The following example deletes the configuration for an interface:

```
{
    "actions": [
        {
            "delete-cfg-path": {
                "path": "interface ethernet-1/3"
            }
        }
    ]
}
```

set-tools-path

Executes an SR Linux **tools** command.

- `path` is the CLI notation for a **tools** command (without the `tools` keyword). Wildcards are not supported; ranges are supported.
- `value` specifies a required value for the **tools** command in the `path`. If the **tools** command does not require a value, the `value` must still be included, but left empty.

The following example uses the `set-tools-path` action to clear statistics for an interface. This action is equivalent to the **tools interface ethernet-1/1 statistics clear** CLI command.

```
{
    "actions": [
        {
            "set-tools-path": {
                "path": "interface ethernet-1/1 statistics clear",
                "value": ""
            }
        },
    ]
}
```

run-script

Issues a command to execute a script.

The command can chain into another Python or bash script, execute a binary, or execute a script in any language for which the system has an interpreter installed.

The script is executed as the user configured to execute event handler operations (see Configuring the user for event handler operations). If a user is not configured to execute event handler operations, they are executed by local SR Linux user `admin`. The working context of the script is the home directory of the user executing the script.

- `cmdline` is the full command to execute the script, including any options. The command is executed in a bash shell. Specify the command as if it were entered at a shell prompt.

The following action executes a script:

```
{
    "actions": [
        {
            "run-script": {
                "cmdline": "/my-script.sh --arg1=val1 -t"
            }
        }
    ]
}
```

reinvoke-with-delay

Re-invokes the script following a specified delay time.

When the actions returned by a script include a value for `reinvoke-with-delay`, the script is automatically re-invoked following the number of milliseconds specified in the value. When the script is re-invoked, it uses the then-current value of all paths.

Format: `reinvoke-with-delay` : <*delay-time*>

<*delay-time*> can be from 1,000 - 300,000 milliseconds (5 minutes).

The following action causes the script to be re-invoked after 5,000 milliseconds:

```
{
    "actions": [
        {
            "reinvoke-with-delay": 5000
        }
    ]
}
```

always-execute

When set to `true` for an action, causes event handler to always process the action, regardless of whether the same action was processed previously.

By default, when a list of actions is processed, event handler checks the list against actions that have been processed previously. If an action is identical to one that was processed previously, event handler does not re-run the duplicate action. For example, if a script returns an action setting an interface `oper-state` to `down`, and a subsequent action sets the `oper-state` for the same interface to `down`, event handler ignores the duplicate action.

If you want to disable this default behavior and run the action regardless of whether the same action has been run previously, set `always-execute` for the action to `true`.

The following action runs a **tools** command that clears statistics for an interface. The `always-execute` setting for the action is `true`, so the command is run each time the action is received from a script.

```
{
    "actions": [
        {
            "set-tools-path": {
                "path": "interface ethernet-1/1 statistics clear",
                "value": "",
                "always-execute": true
        }
    ]
}
```

If `always-execute` is not set to `true`, then the interface statistics are cleared the first time the action is processed, but not if the same action is received again.

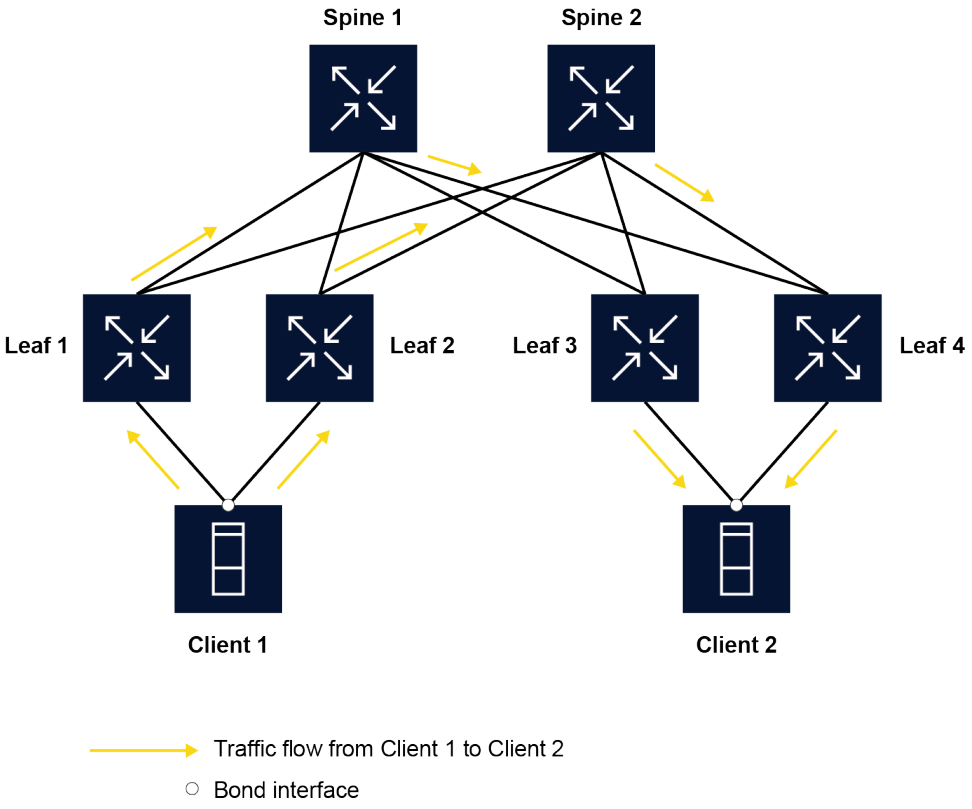
6 Operational groups overview

A potential use for the event handler framework is for operational groups (oper-group for short). The oper-group feature creates a relationship between logical elements of a network node so that they become aware of each other, forming a logical group.

For example, an oper-group can involve a set of downlink ports whose operational state, either up or down, should be configured depending on the operational state of a set of uplink ports. The two sets of ports constitute the oper-group. Event handler can manage the relationship between the sets of ports in the oper-group, changing the operational state of the ports as necessary.

The oper-group feature can address the issue of traffic black-holing when leaves lose all connectivity to the spine layer. Consider the following simplified Clos topology where clients are multi-homed to leaves:

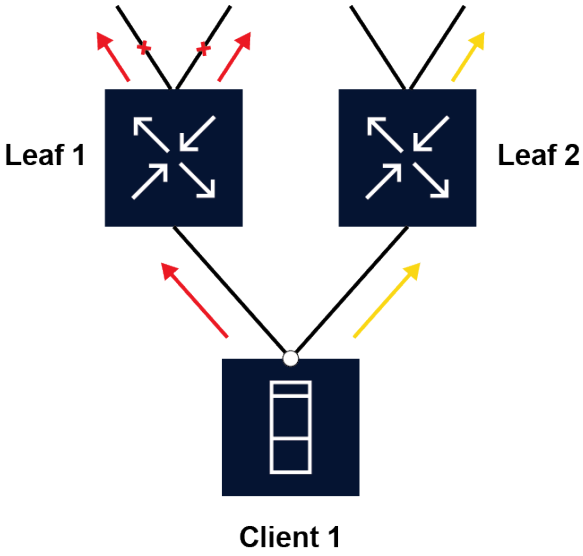
Figure 2: Clos topology with multi-homed clients



With EVPN all-active multihoming enabled in a fabric, traffic from Client 1 is load-balanced over the links attached to the upstream leaves and is propagated via the fabric to its destination.

Because all links of the client's bond interface are active, traffic is hashed to each of the constituent links and therefore uses all available bandwidth. A problem occurs when a leaf loses connectivity to all upstream spines, as illustrated below:

Figure 3: Traffic disruption when a leaf loses connectivity to upstream spines

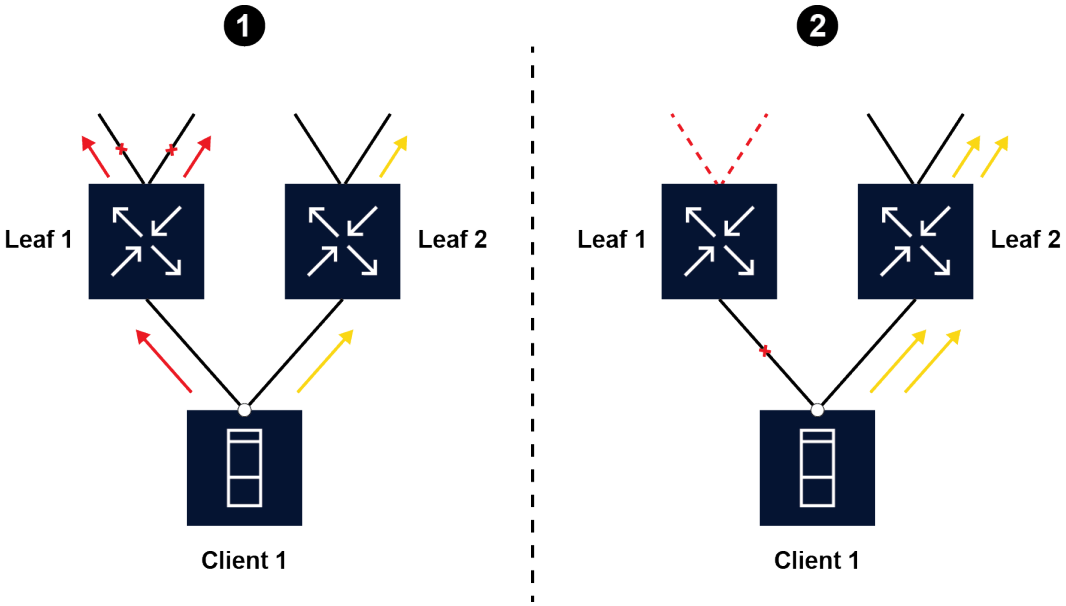


When Leaf 1 loses its uplinks, traffic from Client 1 still flows to Leaf 1 because the client is not aware of any link loss problems happening on the leaf. This results in traffic black-holing on Leaf 1.

An oper-group can remedy this failure scenario by establishing a logical grouping between specific uplink and downlink interfaces on the leaves so that the operational state of the downlinks is tied to the state of the uplinks.

For this example, an oper-group is configured so that leaves shut down their downlink interfaces if they detect that the uplinks are down. This process is shown in the following diagram:

Figure 4: Using an oper-group to prevent traffic black-holing



In this example, the oper-group feature works as follows:

1. When Leaf 1 loses its uplinks, the oper-group is notified and reacts by operationally disabling the access link toward the client.
2. When Leaf 1's downlink transitions to down state, Client 1's bond interface stops using that interface for hashing, and traffic moves over to the healthy links. Client 1 stops sending to Leaf 1, and all traffic flows to Leaf 2.

Configuring event handler for operational groups shows how to configure the event handler framework to enable the oper-group feature.

7 Configuring event handler for operational groups

The key to preventing traffic from being black-holed is to not allow it to be forwarded to a leaf that has no active uplinks; for example, by disabling access links as soon as uplinks become operationally disabled.

The following sections provide an example of configuring event handler for an operational group (oper-group):

- Configuring the event handler instance
- MicroPython script for oper-group
- Displaying oper-group information

7.1 Configuring the event handler instance

To configure an event handler instance for the oper-group feature:

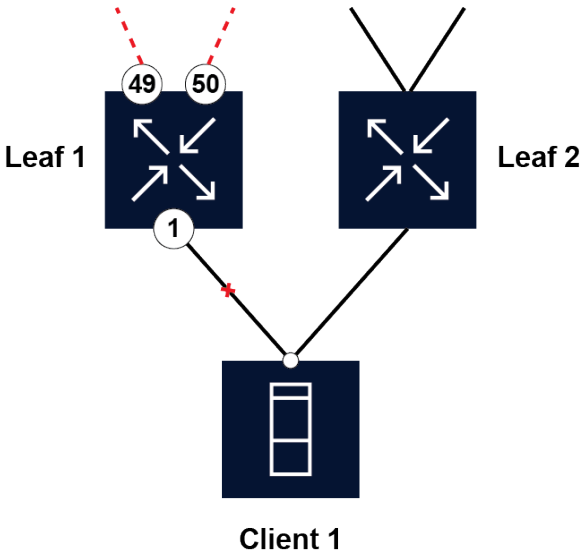
1. Define a set of uplinks to monitor in the `paths` statement.
2. Specify downlinks (access links) and other parameters in the `options` statement.
3. Provide the name of a MicroPython script in the `upython-script` statement.

Defining uplinks to monitor in the `paths` statement

In the `paths` statement of an event handler instance used with an oper-group, you define the set of uplinks that are necessary to provide service for a set of downlinks. The oper-group feature works by monitoring the operational state of the uplinks, and uses the state information to determine whether the operational state of the access links must be changed.

In this example, the operational state of two uplink interfaces `ethernet-1/49` and `ethernet-1/50` on Leaf 1 are being monitored. If the operational state of the uplink interfaces changes to `down`, the oper-group feature changes the state of the access interface to `down` to avoid black-holing of traffic from the client.

Figure 5: Disabling downlink based on uplink state



To monitor the operational state for a set of interfaces, configure the `paths` statement in an event handler instance. For example:

```
--{ candidate shared default }--[  ]--
# info system event-handler instance opergroup path
    system {
        event-handler {
            instance opergroup {
                paths [
                    "interface ethernet-1/{49..50} oper-state"
                ]
            }
        }
    }
```

Specify the contents of the `paths` statement in SR Linux CLI format. In the example above, the `paths` statement is equivalent to the following CLI command:

```
--{ running }--[  ]--
# info from state interface ethernet-1/{49..50} oper-state
```

Specifying downlinks and other parameters in the `options` statement

The `options` statement in the event handler instance allows you to define objects that are passed to the script to be used as input parameters.

For an oper-group configuration, you can use the `options` statement to indicate the relationship between the monitored uplinks and the access links.

In the following example, the options define objects that specify the following:

- The access links that react to state changes of the uplinks
- The number of operationally up uplinks required for the access links to stay up

```
--{ candidate shared default }--[  ]--
```

```
# info system event-handler instance opergroup options
    system {
        event-handler {
            instance opergroup {
                options {
                    object down-links {
                        values [
                            ethernet-1/1
                        ]
                    }
                    object required-up-uplinks {
                        value 1
                    }
                    object debug {
                        value true
                    }
                }
            }
        }
    }
```

In this example, the `down-links` object specifies an interface name. When the object is passed to the script, it can be used as a parameter indicating the access link associated with the uplinks. For this oper-group configuration, the `down-links` object indicates the interface for which the operational state depends on the state of the uplinks defined in the `paths` statement.

The `required-up-uplinks` object specifies the number of uplinks that need to be operationally up before the access link is brought down. For this oper-group configuration, the value is `1`, which means that at least one uplink must be up. The script calculates the number of uplinks that are operationally up, and compares that number to the value in the `required-up-uplinks` object.

The `debug` object is set to `true`, which directs the script to print the values of certain script variables.

Specifying script name in the `upython-script` statement

The `upython-script` statement in the event handler instance specifies the name of the MicroPython script to be invoked when SR Linux detects a change in the interfaces defined in the `paths` statement.

The MicroPython script must reside in one of the following locations:

- `/etc/opt/srlinux/eventmgr` for user-provided scripts
- `/opt/srlinux/eventmgr` for Nokia-provided scripts

```
--{ candidate shared default }--[  ]--
# info system event-handler instance opergroup upython-script
    system {
        event-handler {
            instance opergroup {
                upython-script oper-group.py
            }
        }
    }
```

For this oper-group configuration, whenever a change occurs to the `oper-state` of the interfaces defined in the `paths` statement, event handler invokes the `oper-group.py` script.

Event handler oper-group configuration

When administratively enabled, the full configuration for this event handler instance looks like the following:

```
--{ candidate shared default }--[  ]--
# info system event-handler instance oper-group
    system {
        event-handler {
            instance oper-group {
                admin-state enable
                upython-script oper-group.py
                paths [
                    "interface ethernet-1/{1,2} oper-state"
                ]
                options {
                    object down-links {
                        values [
                            ethernet-1/{20,22}
                        ]
                    }
                    object hold-down-time {
                        value 5000
                    }
                    object required-up-uplinks {
                        value 2
                    }
                }
            }
        }
    }
```

7.2 MicroPython script for oper-group

When there is a state change in any of the paths defined in the `paths` statement of the event handler instance, the script defined in the `upython-script` statement is invoked. Event handler calls the function `event_handler_main()` in the script, passing it a JSON string indicating the current state of the monitored paths, as well as the object:value pairs defined in the `options` statement.

The script receives this input, processes it, and returns a list of actions.

Script input

For the example in Event handler oper-group configuration, the input JSON string consists of the current state of the two uplinks and the provided options. The following JSON string is passed to the `oper-group.py` script if the operational state of interface `ethernet-1/49` changes to `down`:

```
{
    "paths": [
        {
            "path": "interface ethernet-1/49 oper-state",
            "value": "down"
        },
        {
            "path": "interface ethernet-1/50 oper-state",
            "value": "up"
        }
    ],
    "options": {
```

```
        "debug": "true",
        "required-up-uplinks": "1",
        "down-links": [
            "ethernet-1/1"
        ]
    }
}
```

Script processing

The following is the `oper-group.py` script referenced in the event handler instance.

```
import sys
import json

def count_up_uplinks(paths):
    up_cnt = 0
    for path in paths:
        if path.get('value','down') == 'up':
            up_cnt = up_cnt+1
    return up_cnt

def required_up_uplinks(options):
    return int(options.get('required-up-uplinks', '1'))

def hold_time(options):
    return int(options.get('hold-down-time', '0'))

def bool_to_oper_state(val):
    return ('down','up')[bool(val)]

def event_handler_main(in_json_str):
    in_json = json.loads(in_json_str)
    paths = in_json['paths']
    options = in_json['options']
    persist = in_json.get('persistent-data', {})

    num_up_uplinks = count_up_uplinks(paths)
    downlink_should_be_up = required_up_uplinks(options) <= num_up_uplinks
    needs_hold_down = False

    # down->up transition will be held for optional hold-time
    if (hold_time(options) > 0) and downlink_should_be_up:
        needs_hold_down = persist.get("last-state", "up") == "down"

    if options.get("debug") == "true":
        print(
            f"hold down time = {hold_time(options)}ms\n\
num of required up uplinks = {required_up_uplinks(options)}\n\
detected num of up uplinks = {num_up_uplinks}\n\
downlinks new state = {bool_to_oper_state(downlink_should_be_up)}\n\
needs_hold_down = {str(needs_hold_down)}"
        )

    response_actions = []

    oper_state_str = bool_to_oper_state(not needs_hold_down and downlink_should_be_up)
    for downlink in options.get('down-links'):
        response_actions.append({'set-ephemeral-path' : {'path':'interface {0} oper-
state'.format(downlink),'value':oper_state_str}})

    if needs_hold_down:
        response_actions.append({'reinvoke-with-delay' : hold_time(options)})
```

```
    response_persistent_data = {'last-state':bool_to_oper_state(downlink_should_be_up)}

    response = {'actions':response_actions,'persistent-data':response_persistent_data}
    return json.dumps(response)
```

The following sections describe how each part the script processes the input for this oper-group example.

Parsing input JSON

Starting with the `event_handler_main` function, the incoming JSON string is parsed and the relevant portions are extracted.

```
def event_handler_main(in_json_str):
    in_json = json.loads(in_json_str)
    paths = in_json['paths']
    options = in_json['options']
    persist = in_json.get('persistent-data', {})
```

The `paths` and `options` are objects defined in the incoming JSON string, and they are saved in their respective like-named variables.

Determing the state for the downlink

With the input parsed, the script determines the required state of the downlink, based on the received input.

```
    num_up_uplinks = count_up_uplinks(paths)
    downlink_should_be_up = required_up_uplinks(options) <= num_up_uplinks
    needs_hold_down = False
```

First, the script counts the number of uplinks in `oper-state up` using the `count_up_uplinks()` function, which simply walks through the current state of the uplinks passed into the script by event handler.

```
def count_up_uplinks(paths):
    up_cnt = 0
    for path in paths:
        if path.get('value','down') == 'up':
            up_cnt = up_cnt+1
    return up_cnt
```

After calculating how many uplinks are operationally up, the script determines the required state for the downlinks. To make this decision, it compares the number of operational uplinks to the required number of uplinks (from the `required-up-uplinks` option):

If the required number of operationally up uplinks is less than the required number, the downlink is set operationally down to prevent traffic black-holing. On the other hand, if the number of operational uplinks is greater than or equal to the required number of uplinks, the downlink is set operationally up.

The calculated state of the downlink is saved in the `downlinks_new_state` variable.

Populating the debug log

The `debug` option causes the script variables to appear in the debug log.

```
    if options.get("debug") == "true":
        print(
            f"hold down time = {hold_time(options)}ms\n\
```

```
num of required up uplinks = {required_up_uplinks(options)}\n\
detected num of up uplinks = {num_up_uplinks}\n\
downlinks new state = {bool_to_oper_state(downlink_should_be_up)}\n\
needs_hold_down = {str(needs_hold_down)}"
        )
```

The debug log is present only if the `debug` option is set to `"true"` in the event handler instance configuration.

You can display the debug log by using the following CLI command:

```
--{ running }--[  ]--
# info from state system event-handler instance opergroup last-stdout-stderr
```

Composing output

At this point, the script is able to define the correct state for the downlinks, based on the state of the monitored uplinks and the required number of healthy uplinks. For the event handler to take action, the script needs to output a JSON string following the format defined in Actions.

```
    response_actions = []

    oper_state_str = bool_to_oper_state(not needs_hold_down and downlink_should_be_up)
    for downlink in options.get('down-links'):
        response_actions.append({'set-ephemeral-path' : {'path':'interface {0} oper-
state'.format(downlink),'value':oper_state_str}})

    if needs_hold_down:
        response_actions.append({'reinvoke-with-delay' : hold_time(options)})
    response_persistent_data = {'last-state':bool_to_oper_state(downlink_should_be_up)}

    response = {'actions':response_actions,'persistent-data':response_persistent_data}
    return json.dumps(response)
```

This example shows an output JSON string, using the calculated `downlinks_new_state` and the list of downlinks provided from the `down-links` option.

The output JSON string contains the set-ephemeral-path action, which sets the `oper-state` of the downlink to the correct value (`up` or `down`).

The output is provided via the `response` dictionary, and is JSON-encoded before returning from the function. This routine provides a JSON string back to the event handler, which processes and executes the actions passed to it.

The result of this processing shows the implementation of the oper-group feature: the event handler executes actions to set the state of a downlink based on the state of a group of uplinks.

7.3 Displaying oper-group information

When an event handler instance is configured and administratively enabled, an initial sync of the monitored paths state is performed. As a result of this initial sync, event handler immediately attempts to execute a script when it receives the state for the monitored paths.

You can display the status of an event handler instance by querying the state datastore. For example:

```
# /info from state system event-handler instance opergrp
```

```
    system {
        event-handler {
            instance opergrp {
                admin-state enable
                upython-script oper-group.py
                oper-state up
                paths [
                    "interface ethernet-1/1 oper-state"
                    "interface ethernet-1/4 oper-state"
                ]
                options {
                    object down-links {
                        values [
                            ethernet-1/3
                            ethernet-1/8
                        ]
                    }
                    object required-num-up-links {
                        value 2
                    }
                }
                last-execution {
                    start-time now
                    end-time now
                    upython-duration 1
                    input "{\"paths\":[{\"path\":\"interface ethernet-1/1 oper-state\",\
"value\":\"up\"},{\"path\":\"interface ethernet-1/4 oper-state\",\"value\":\"up\"}],\"options
\":{\"down-links\":[\"ethernet-1/3\",\"ethernet-1/8\"],\"required-num-up-links\":\"2\"},\
"persistent-data\":{\"last-state\":\"up\"}}"
                    output "{\"actions\": [{\"set-ephemeral-path\": {\"path\": \"interface
 ethernet-1/3 oper-state\", \"value\": \"up\"}}, {\"set-ephemeral-path\": {\"path\": \
"interface ethernet-1/8 oper-state\", \"value\": \"up\"}}], \"persistent-data\": {\"last-state\
": \"up\"}}"
                    stdout-stderr ""
                }
                last-errored-execution {
                    oper-down-reason admin-disabled
                    oper-down-reason-detail ""
                    start-time "26 seconds ago"
                    end-time "25 seconds ago"
                    upython-duration 0
                    input "{\"paths\":[{\"path\":\"interface ethernet-1/1 oper-state\",\"value
\":\"up\"},{\"path\":\"interface ethernet-1/4 oper-state\",\"value\":\"down\"}],\"options
\":{\"down-links\":[\"ethernet-1/3\",\"ethernet-1/8\"],\"required-num-up-links\":\"2\"},\
"persistent-data\":{\"last-state\":\"down\"}}"
                    output "{\"actions\": [{\"set-ephemeral-path\": {\"path\": \"interface
 ethernet-1/3 oper-state\", \"value\": \"up\"}}, {\"set-ephemeral-path\": {\"path\": \
"interface ethernet-1/8 oper-state\", \"value\": \"up\"}}], \"persistent-data\": {\"last-state\
": \"up\"}}"
                    stdout-stderr ""
                }
                statistics {
                    upython-duration 516
                    execution-count 1643
                    execution-successes 1642
                    execution-errors 1
                }
            }
        }
    }
```

This command displays the following information:

- `oper-state`

The operational state of the event handler instance. In case of any errors in the script and, or configuration the state is `down`.

- `last-execution`
  Information about the most recent time the script was executed.
- `last-errored-execution`
  Information about the last time the script was executed with an error result. This includes the `oper-down-reason`, `oper-down-reason-detail` the input and output JSON strings, and the output print statements and log messages sent to `stdout-stderr` by the script.
- `statistics`
  Statistics related to the execution process.

For the oper-group example, the following output is displayed if one of the uplinks goes down:

```
--{ running }--[  ]--
# info from state system event-handler instance opergrp
    system {
        event-handler {
            instance opergrp {
                admin-state enable
                upython-script oper-group.py
                oper-state up
                last-execution {
                    start-time now
                    end-time now
                    upython-duration 1
                    input "{\"paths\":[{\"path\":\"interface ethernet-1/1 oper-state\",\
"value\":\"up\"},{\"path\":\"interface ethernet-1/4 oper-state\",\"value\":\"up\"}],\"options
\":{\"down-links\":[\"ethernet-1/3\",\"ethernet-1/8\"],\"required-num-up-links\":\"2\"},\
"persistent-data\":{\"last-state\":\"up\"}}"
                    output "{\"actions\": [{\"set-ephemeral-path\": {\"path\": \"interface
 ethernet-1/3 oper-state\", \"value\": \"up\"}}, {\"set-ephemeral-path\": {\"path\": \
"interface ethernet-1/8 oper-state\", \"value\": \"up\"}}], \"persistent-data\": {\"last-state\
": \"up\"}}"
                    stdout-stderr ""num of required up uplinks = 1
detected num of up uplinks = 1
downlinks new state = up"
                }
            }
        }
    }
```

The setting for `downlinks new state` is `up` because the `detected num of up uplinks` did not drop below the required number of 1. The downlink interface therefore remains operationally up.

If both of the uplinks go down, the following is displayed:

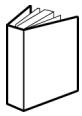
```
--{ running }--[  ]--
# info from state system event-handler instance opergrp
# info from state system event-handler instance opergrp
    system {
        event-handler {
            instance opergrp {
                admin-state enable
                upython-script oper-group.py
                oper-state up
                last-execution {
                    start-time now
                    end-time now
                    upython-duration 1
```

```
                    input "{\"paths\":[{\"path\":\"interface ethernet-1/1 oper-state\",\
"value\":\"up\"},{\"path\":\"interface ethernet-1/4 oper-state\",\"value\":\"up\"}],\"options
\":{\"down-links\":[\"ethernet-1/3\",\"ethernet-1/8\"],\"required-num-up-links\":\"2\"},\
"persistent-data\":{\"last-state\":\"up\"}}"
                    output "{\"actions\": [{\"set-ephemeral-path\": {\"path\": \"interface
 ethernet-1/3 oper-state\", \"value\": \"up\"}}, {\"set-ephemeral-path\": {\"path\": \
"interface ethernet-1/8 oper-state\", \"value\": \"up\"}}], \"persistent-data\": {\"last-state\
": \"up\"}}"
                    stdout-stderr ""num of required up uplinks = 1
detected num of up uplinks = 0
downlinks new state = down"
                }
            }
        }
    }
```

The `detected num of up uplinks` is 0, which is below the required number of 1. This causes event handler to set the downlink interface to operationally down.

In this way, event handler uses the oper-group feature to disable the access link when the uplink interfaces go down, therefore preventing traffic from black-holing.

Customer document and product support



Customer documentation

Customer documentation welcome page

Technical support

Product support portal

Documentation feedback

Customer documentation feedback