Switch Management API

Version 21.1.1, 29 January 2021

Registered Address	Support	Sales
26, Kingston Terrace, Princeton, New Jersey 08540, United States		
		+91 80 4850 5445
http://www.rtbrick.com	support@rtbrick.com	sales@rtbrick.com

Table of Contents

Abstract

This document outlines the basic components which are deployed on the switch for managing the switch and their interaction to other systems. It gives an overview of the different types of APIs and outlines the usage of that APIs.

1. Overview

A fabric switch uses different protocols to interface with the control and management plane and vice versa. This section gives a brief overview of the communication flow. More detailed information will be provided in the next sections and in dedicated documents.

Zero-Touch Provisioning (ZTP) is driven by DHCP options. First the Open Network Installation Environment (ONIE) that is installed on a bare metal switch by default, discovers the RtBrick Full Stack (RBFS) installer image and installs RBFS /ONIE/. ONIE supports HTTP and TFTP to load the installer image. After the first reboot the switch loads the startup configuration. The base URL is read from DHCP option 210 and defaults to the DHCP server if DHCP option 210 is not set /ZTP/. ONIE has only access to the dedicated management interface. Consequently ZTP requires out-of-band access to the DHCP service and to download the RBFS installer image and the startup configuration.

RADIUS messages are used to provision subscriber sessions. All per-subscriber settings are encoded in RADIUS attributes /RADIUS/. RADIUS messages are sent in-band.

Switch-global configuration like credentials, BGP peerings or CoS profiles for example, are either part of the startup configuration files or applied through REST API invocations.

The switch management high-level communication flow and provided APIs are illustrated below:

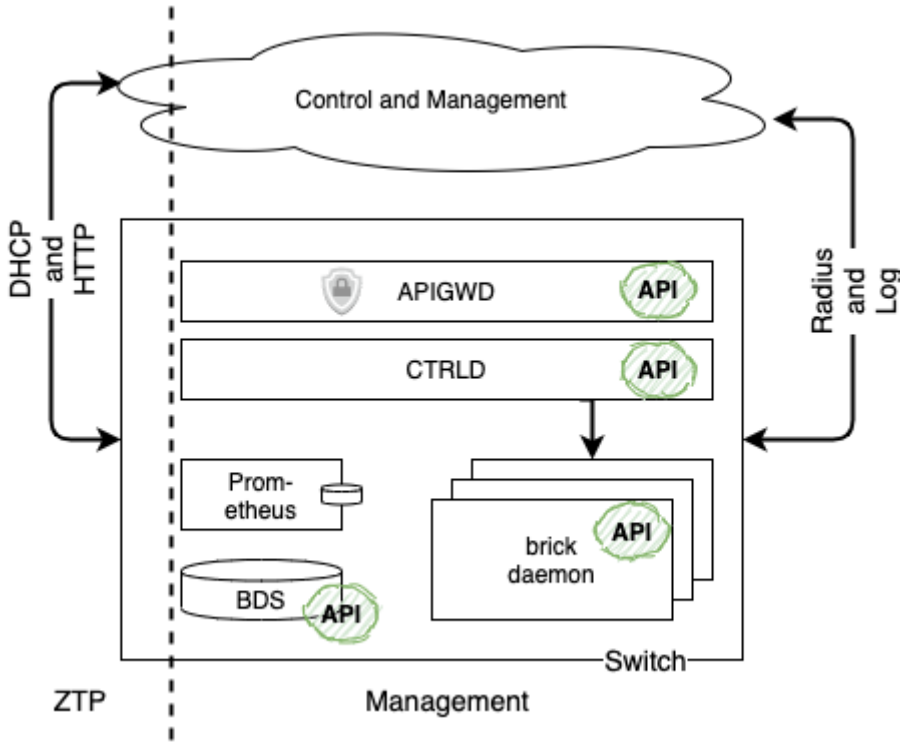

Figure 1. Switch Management APIs

Most of the installed components on the switch provide an API.

The API Gateway Daemon (APIGWD) is the TLS endpoint for the switch and converts an external access token into an internal rtbrick token /SEC/. Further it forwards the requests to Control Daemon (CTRLD).

The CTRLD is the entry point to RBFS. In essence it provides a proxy to the other APIs and also higher level APIs for switch configuration. Apart from that it handles the ZTP post process.

The Brick Data Store (BDS) provides a really Low-level API to the BDS tables and objects. Every Brick Daemon (BD) can provide additional APIs beside the BDS API. Additionally to these REST APIs also parts of the Radius protocol are implemented.

2. RBFS API Fundamentals

2.1. RBFS in a Nutshell

RtBrick Full Stack (RBFS) architecture basics and an understanding of the RBFS API characteristics is needed to make use of the RBFS APIs. The Brick Data Store (BDS) is a schema-driven in-memory database optimized for networking use cases. It stores objects, which are described in schemas, and organizes them in different tables. Each brick daemon has a single responsibility and defines the tables and schemas needed for its respective task. Brick daemons interact with each other over a pub-sub facility provided by the BDS.

RBFS differentiates between persistent and non-persistent configuration. Non-persistent configuration is lost after each switch-reboot whereas persistent configuration is written to disk. Service related information is an example for non-persistent configuration. All service information is lost when a switch is rebooted and has to be pushed again on demand by the controller after the reboot. The startup configuration is an example for persistent data. Whenever startup related configuration gets changed, e.g. by adding a new BGP peering, the startup configuration file gets updated to preserve the new settings if the switch reboots.

2.2. Low-level BDS API

The BDS API provides access to the BDS tables and objects. The BDS API relies on RPC over HTTP and defines an envelope protocol to convey BDS objects. The data exchange format is JSON.

The BDS API supports adding objects, removing objects and walking through a table. Working with BDS tables and objects requires awareness of the indices that exist on each table. The add operation adds new or replaces existing objects.

The BDS API only validates syntactic but not semantic integrity.

The delete operation removes objects. A merge operation to add and remove objects in one go does not exist. Walking a table requires to specify the index the walk operation should follow. The sequence index is used by default if no index is specified. The sequence index represents the temporal insertion order of objects into a table. The objects are returned in the index sort order. Searching for objects requires an understanding of key and non-key attributes of each index.

The usage of the BDS API also requires an understanding of the relations between the objects. It can be compared to referential integrity in SQL databases which also forces the client to modify database records in the right order. Using the BDS API for integration is as if you would use SQL (or the database model) rather than an API of a software product relying on a relational database.

The BDS schema definitions get constantly honed and can become subject of non-backward compatible changes.

For example, BDS schemas definitions can be modified to simplify CLI implementation without changing the data that is being exchanged as such. Even a patch can introduce a non-backward compatible BDS schema definition change.

For every brick daemon there is a dedicated BDS API endpoint.

2.3. Brick Daemon API

Every brick daemon provides additional APIs beside the BDS API. The configuration daemon is a good example. It introduces a transactional configuration API that allows you to incrementally update a candidate configuration that is later applied in one go. The configuration daemon detects deltas between configurations and only the detected deltas trigger subsequent actions. For example, if a new BGP peer has been added to the candidate config, the additional BGP peering gets established without resetting established peerings.

The configuration daemon processes the persistent configuration.

All configurations applied via config daemon will be added to the startup configuration file and applied after every reboot. Not all BDS tables become part of the persistent configuration by default. A brick daemon can declare a BDS table persistent. In this case the ownership of the table is transferred from the respective brick daemon to the configuration daemon. The brick daemon cannot modify the objects in BDS tables when the ownership has been transferred to configuration daemon. Consequently the BDS low-level API can still be used for reading objects but not for object modifications. All changes are applied by the configuration daemon. The configuration daemon provides a delta processing and the brick daemons get notified about configuration changes via the BDS pub-sub facility mentioned earlier.

2.4. CTRLD API

The control daemon (CTRLD) is the entry point to RBFS. In addition, CTRLD implements the ZTP startup configuration discovery by means of downloading the startup configuration files from the retrieved base URL and applying it to the switch.

CTRLD exposes a REST-API to program the switch, which is kept backward compatible /CTRLD/. CTRLD itself uses the low-level BDS API as needed. For example, CTRLD API supports to specify which metric shall be sampled and to define alert conditions which are then stored in BDS tables. CTRLD takes care of compatibility issues outlined earlier. RBFS leverages Prometheus for metric sampling. All counters and metrics of interest shall be sampled by Prometheus and

accessed through PromQL rather than the BDS API. A local history of all sampled metrics is also very handy for troubleshooting.

The CTRLD API also provides access to the low-level BDS API endpoints. The usage of the BDS API is an indication that a CTRLD API endpoint is missing. It is recommended to define an external representation of state implemented by the CTRLD API rather than extensively working with the BDS objects in order to avoid compatibility issues. Since CTRLD runs on the switch it needs only to support the current BDS schema definition to implicitly provide backward compatibility.

2.5. RADIUS

The subscriber session establishment relies on RADIUS. All per-subscriber settings are exchanged through RADIUS messages whereas information that is shared across multiple subscribers, like QoS profiles, need to be configured through API invocations.

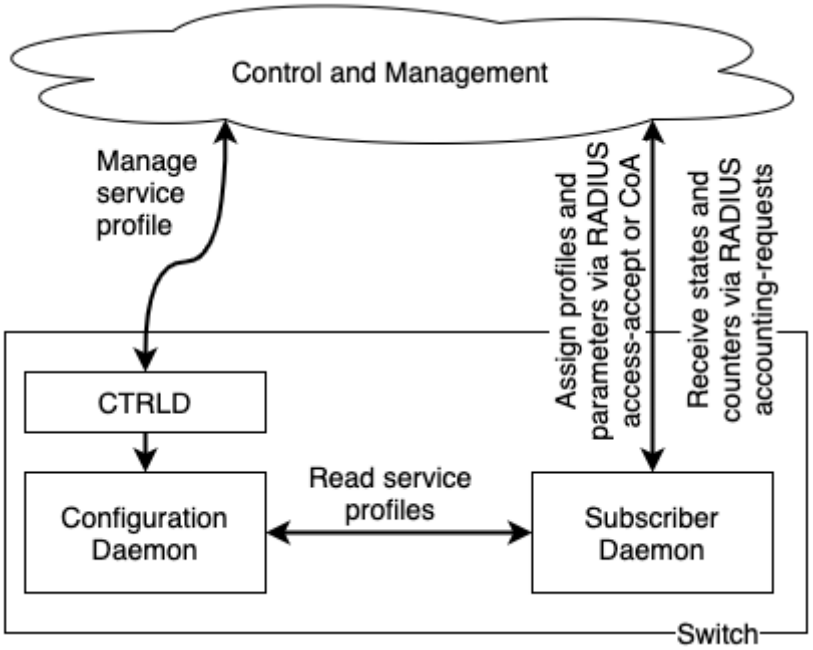

Figure 2. Overview of REST API and RADIUS calls for subscriber management

Subscriber session settings are non-persistent and are lost if a switch reboots or the session ends. The sessions must be established again after each reboot. RADIUS relies on UDP for maximum throughput and minimum latency. Message loss is handled by the application rather than the transport layer. The application asks for the message again if the message is not delivered in time. In addition the application can cope with receiving the same message multiple times.

2.6. Summary

The previous sections provided a quick overview of the existing RBFS APIs and what they are intended for:

- Use the CTRLD API to maintain the persistent configuration. All settings that shall preserve a switch reboot need to become part of the persistent configuration. Running configuration supports a delta processing and only identified deltas trigger an action on the switch. QoS profiles are a good example for persistent configurations.
- Use the CTRLD API to configure sampling of sensors and counters. Use PromQL, the prometheus query language, to access the sampled values. Additionally program alert conditions as needed. All alerts are sent to a GELF log infrastructure /GELF/.
- Use RADIUS messages to exchange per-subscriber settings for session establishment.
- Be aware that BDS schemas can change when using the low-level BDS API.

3. Configurations

This sections describes the configuration files of the APIGWD and CTRLD.

3.1. APIGWD

In a production environment, the APIGWD binary starts with default parameters. This service of APIGWD is called rtbrick-apigwd.

To see the default parameters and therefore also the files where configurations are stored, the easiest way to figure out is to do a apigwd -help.

To see the actual installed APIGWD version we can use apigwd -version.

APIGWD has in essence 3 important configuration files:

- /etc/rtbrick/apigwd/config.json: Configuration for the apigwd
- /etc/rtbrick/apigwd/access_secret_jwks.json: JWKS file for external communication
- /etc/rtbrick/apigwd/tls.pem: X509 public/private key file in pem format

3.1.1. config.json

This section describes the main configuration file of APIGWD. This file can be changed on the file system, APIGWD has a file watcher on the file, and does a reload when the file changes.

/etc/rtbrick/apigwd/config.json example

```
{
  "oidc_issuer" : "http://192.168.202.56:8080/auth/realms/RBMS",
  "client_id" : "rtbrick-switch",
  "client_secret" : "17863a7b-ddf3-432e-8392-1e3815eb64d2",
  "redirect_url": "",

  "request_rate": 5,
  "request_burst": 10,
  "report_rejects_every": 10
}
```

Table 1. /etc/rtbrick/apigwd/config.json format

Name	Type	Description
oidc_issuer	string	OIDC issuer url
client_id	string	OIDC client ID

Name	Type	Description
client_secret	string	OIDC client secret
redirect_url	string	OIDC RedirectURL
request_rate	float	The allowed requests per second per client.
request_burst	int	Is the maximum number of tokens that can be consumed at once, without respect the rate.
report_rejects_every	int	Report rejects only every x seconds, to avoid massive logging to a gelf endpoint.

3.1.2. access_secret_jwks.json

This section describes the access_secret_jwks.json file. This file can be changed on the file system, APIGWD has a file watcher on the file, and does a reload when the file changes.

JSON Web Key Set (JKWS) is described in the RFC 7517.

/etc/rtbrick/apigwd/access_secret_jwks.json example

```
{
  "keys": [
    {
      "kty": "RSA",
      "e": "AQAB",
      "use": "sig",
      "kid": "access",
      "alg": "RS256",
      "n": "NOT A REAL KEY"
    }
  ]
}
```

This keys are for authentication of external call towards to the APIGWD.

The right key is selected by the kid (key id). With this key the access tokens are verified and converted to a rtbrick token.

3.1.3. tls.pem

This section describes the tls.pem file. This file contains the TLS certificate (public and private key) used to serve the TLS endpoint.

If the file is not specified a new self signed certificate is created.

This file can be changed on the file system, APIGWD has a file watcher on the file, and does a reload when the file changes.

This file is an X509 public/private key file in PEM format specified in the RFC7468 .

/etc/rtbrick/apigwd/tls.pem example

```
-----BEGIN CERTIFICATE-----
NOT A REAL KEY
-----END CERTIFICATE-----
-----BEGIN RSA PRIVATE KEY-----
NOT A REAL KEY
-----END RSA PRIVATE KEY-----
```

3.2. CTRLD

In a production environment, the CTRLD binary starts with default parameters. This service of CTRLD is called rtbrick-ctrld.

To see the default parameters and therefore also the files where configurations are stored, the easiest way to figure out is to do a ctrld -help.

To see the actual installed CTRLD version we can use ctrld -version.

CRTLD has in essence 3 important configuration files:

- /etc/rtbrick/ctrld/config.json: Configuration for the ctrld
- /etc/rtbrick/ctrld/policy.json: Role Based Access Control policy file.
- /var/lib/lxc/<container-name>/element.config: Element configuration file per container.

3.2.1. config.json

This section describes the main configuration file of CTRLD. This file can be changed via API, if it is changed on the file system CTRLD has to be restarted.

/etc/rtbrick/ctrld/config.json example

```
{
  "rbms_enable": true,
  "rbms_host": "http://10.200.32.48",
  "rbms_authorization_header": "Bearer THIS IS NOT A REAL KEY",
  "rbms_heart_beat_interval": 10,
  "graylog_enable": true,
  "graylog_url": "http://10.200.32.49:12201/gelf",
  "graylog_heart_beat_interval": 10,
  "graylog_endpoints": [
    {
      "name": "ztp",
      "url": "http://192.168.202.46:12201/gelf"
    }
  ],
  "auth_enabled": false
}
```

Table 2. /etc/rtbrick/ctrld/config.json format

Name	Type	Description
rbms_enable	bool	To enable all RBMS outgoing messages rbms_host
rbms_host	string	RBMS base url e.g.: http://192.168.202.44:9009
rbms_authorization_header	string	RBMS Authorization Header is set to all calls which are outgoing to RBMS
rbms_heart_beat_interval	int	RBMS heartbeat Interval in seconds (0 means deactivated)
graylog_enable	bool	To Enable all Graylog outgoing messages
graylog_url	string	Graylog url e.g. http://127.0.0.1:12201/gelf
graylog_heart_beat_interval	string	Graylog heartbeat Interval in seconds (0 means deactivated)

<table>
<tr><th>Name</th><th>Type</th><th>Description</th></tr>
<tr><td>graylog_endpoints</td><td colspan="2">GraylogEndpoints allows to specify multiple graylog endpoints by name. If a log to a specific endpoint is requested and the endpoint is not available, the log is send to the default graylog endpoint.

<table>
<tr><th>Name</th><th>Type</th><th>Description</th></tr>
<tr><td>name</td><td>string</td><td>Logical name of the entpoint e.g.: ztp for ztp messages.</td></tr>
<tr><td>url</td><td>string</td><td>Graylog url e.g. http://127.0.0.1:12201/gelf</td></tr>
<tr><td>disable</td><td>string</td><td>Disables this endpoint.</td></tr>
</table>

If the default endpoint is disabled, but the specific one is enabled than the message to the specific endpoint will be sent.

If the default endpoint is enabled, but the specific one is disabled than the message to the specific endpoint will not be sent.</td></tr>
<tr><td>auth_enabled</td><td>bool</td><td>To Enable the authorization and authentication</td></tr>
</table>

3.2.2. policy.json

This section shows the **r**ole **b**ased **a**ccess **c**ontrol (RBAC) configuration for CTRLD. This file can be changed via API, if it is changed on the file system CTRLD has to be restarted.

/etc/rtbrick/ctrld/policy.json example

```
{
  "permissions": [
    {"sub": "system", "obj": "/*", "act": ".*" },
    {"sub": "supervisor", "obj": "/*", "act": ".*" },
    {"sub": "operator", "obj": "/*", "act": ".*"},
    {"sub": "reader", "obj": "/*", "act": "GET"},
    {"sub": "reader", "obj":
"/api/v1/rbfs/elements/{element_name}/services/{service_name}/proxy/bds/table
/walk", "act": ".*"},
    {"sub": "reader", "obj":
"/api/v1/rbfs/elements/{element_name}/services/{service_name}/proxy/bds/objec
t/get", "act": ".*"}
  ]
}
```

Table 3. /etc/rtbrick/ctrld/policy.json format

Name	Type	Description
sub	string	Subjects means the role which has the permission. Here RegexMatch Function is used: a regular expression pattern matcher.
obj	string	Object is the REST endpoint. Here KeyMatch4 Function is used: KeyMatch4 determines whether key1 matches the pattern of key2 (similar to RESTful path), key2 can contain a * and other patterns: • "/foo/bar" matches "/foo/*" • "/resource1" matches "/{resource}" • "/parent/123/child/123" matches "/parent/{id}/child/{id}" • "/parent/123/child/456" does not match "/parent/{id}/child/{id}"
act	string	And Action is the HTTP Method. Here RegexMatch Function is used: a regular expression pattern matcher.

So the example rules mean:

- The user with the role system is allowed to access all rest endpoints, and act on them with all HTTP methods.
- The user with the role reader is allowed to access all rest endpoints, but can only call the HTTP GET method.
- All authenticated users are allowed to access the proxy endpoint with all HTTP methods.The user with the role system is allowed to access all rest endpoints, and act on them with all HTTP methods.

3.2.3. element.config

This section shows the element.config which can be created per container. This files allows to redefine the element name, so the name can defer from the container name.

/var/lib/lxc/<container-name>/element.config example

```
{
  "element_name": "l1.pod1.blr",
  "pod_name": "blr",
  "ztp_enabled": false
}
```

Table 4. /var/lib/lxc/<container-name>/element.config format

Name	Type	Description
element_name	string	Name of the Element (Default container name)
pod_name	string	Name of the POD
ztp_enabled	bool	If enabled the ztp post process starts after the switch changes to operational state up. It's recommended to set this to false. In that case only the initial installation or reinstallation will trigger that process.

Unresolved directive in switch_mgmt_api_ref.adoc - include::src/04_switch_mgmt_related_documentation.adoc[]