ZTPServer Documentation

Release 1.5.0

Arista Networks

September 27, 2017

Contents

ZTPServer provides a bootstrap environment for Arista EOS based products. It is written mostly in Python and leverages standard protocols like DHCP (for boot functions), HTTP (for bi-directional transport), XMPP and syslog (for logging). Most of the configuration files are YAML based.

This open source project is maintained by the Arista Networks EOS+ services organization.

CHAPTER 1

Highlights

- Extends the basic capability of EOS's zero-touch provisioning feature in order to allow more robust provisioning activities
- Is extensible, for easy integration into various network environments
- Can be run natively in EOS or any Linux server
- Arista EOS+ led community open source project

CHAPTER 2

Features

- Dynamic startup-config generation and automatic install
- Image and file system validation and standardization
- Connectivity validation and topology based auto-provisioning
- Config and device templates with dynamic resource allocation
- Zero-touch replacement and upgrade capabilities
- User extensible actions
- Email, XMPP, syslog based

Overview

ZTPServer provides a robust server which enables comprehensive bootstrap solutions for Arista network elements. ZTPserver takes advantage of the the ZeroTouch Provisioning (ZTP) feature in Arista's EOS (Extensible Operating System) which enables a node to connect to a provisioning server whenever a valid configuration file is missing from the internal flash storage.

ZTPServer provides a number of features that extend beyond simply loading a configuration file and a boot image on a node, including:

- sending an advanced bootstrap client to the node
- mapping each node to an individual definition which describes the bootstrap steps specific to that node
- defining configuration templates and actions which can be shared by multiple nodes - the actions can be customised using statically-defined or dynamically-generated attributes
- implementing environment-specific actions which integrate with external/internal management systems
- validation topology using a simple syntax for expressing LLDP neighbor adjacencies
- enabling Zero Touch Replacement, as well as configuration backup and management

ZTPServer is written in Python and leverages standard protocols like DHCP (DHCP options for boot functions), HTTP(S) (for bi-directional transport), XMPP and syslog (for logging). Most of the configuration files are YAML-based.

Highlights:

- extends the basic capability of ZTP (in EOS) to allow more robust provisioning activities
- is extensible and easy to integrate into any operational environment
- can be run natively in EOS or on a separate server
- is developed by a community lead by Arista's EOS+ team as an open-source project

Features:

- automated configuration file generation
- image and file system validation and standardization
- cable and connectivity validation
- topology-based auto-provisioning
- configuration templating with resource allocation (for dynamic deployments)
- Zero Touch Replacement and software upgrade capabilities
- user extensible actions
- XMPP and syslog-based logging and accounting

ZTP Intro

Zero Touch Provisioning (ZTP) is a feature in Arista EOS's which, in the absence of a valid startup-config file, enables nodes to be configured over the networks.

The basic flow is as follows:

- check for startup-config, if absent, enter ZTP mode
- send DHCP requests on all connected interfaces
- if a DHCP response is received with Option 67 defined (bootfile-name), retrieve that file
- if that file is a startup-config, then save it to startup-config and reboot
- if that file is an executable, then execute it. Common actions executed this way include upgrading the EOS image, downloading extension packages, and dynamically building a startup-config file. (**ZTPServer's bootstrap script is launched this way**)
- reboot with the new configuration

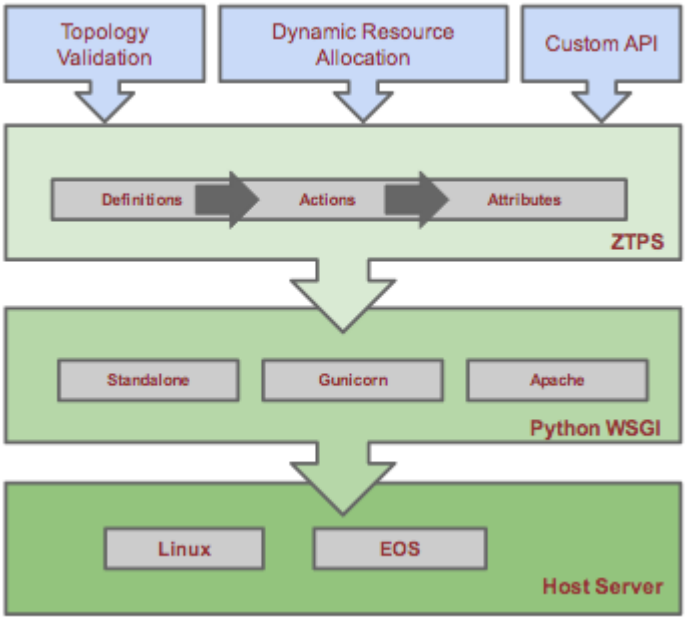
See the ZTP Tech Bulletin and the Press Release for more information on ZTP.

Architecture

There are 2 primary components of the ZTPServer implementation:

- the **server** or ZTPServer instance **AND**
- the **client** or bootstrap (a process running on each node, which connects back to the server in order to provision the node)

Server



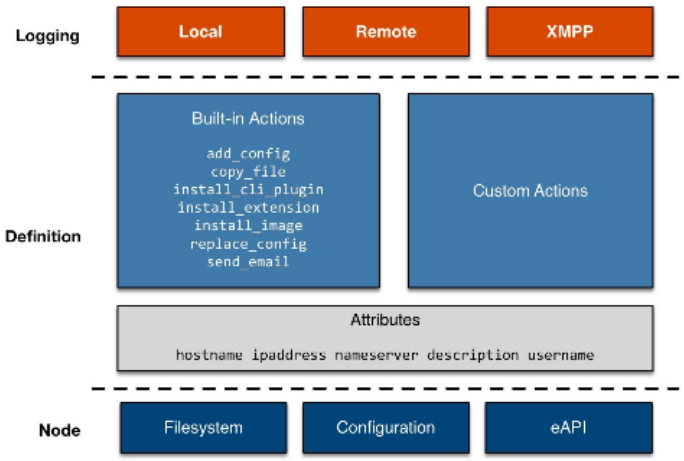
The server can run on any standard x86 server. Currently the only OS-es tested are Linux and MacOS, but theoretically any system that supports Python could run ZTPServer. The server provides a Python WSGI compliant interface, along with a standalone HTTP server. The built-in HTTP server runs by default on port 8080 and provides bidirectional file transport and communication for the bootstrap process.

The primary methods of provisioning a node are:

- **statically** via mappings between node IDs (serial number or system MAC address) and configuration definitions OR
- **dynamically** via mapping between topology information (LLDP neighbors) and configuration definitions

The definitions associated with the nodes contain a set of actions that can perform a variety of functions that ultimately lead to a final device configuration. Actions can use statically configured attributes or leverage configuration templates and dynamically allocated resources (via resource pools) in order to generate the system configuration. Definitions, actions, attributes, templates, and resources are all defined in YAML files.

Client



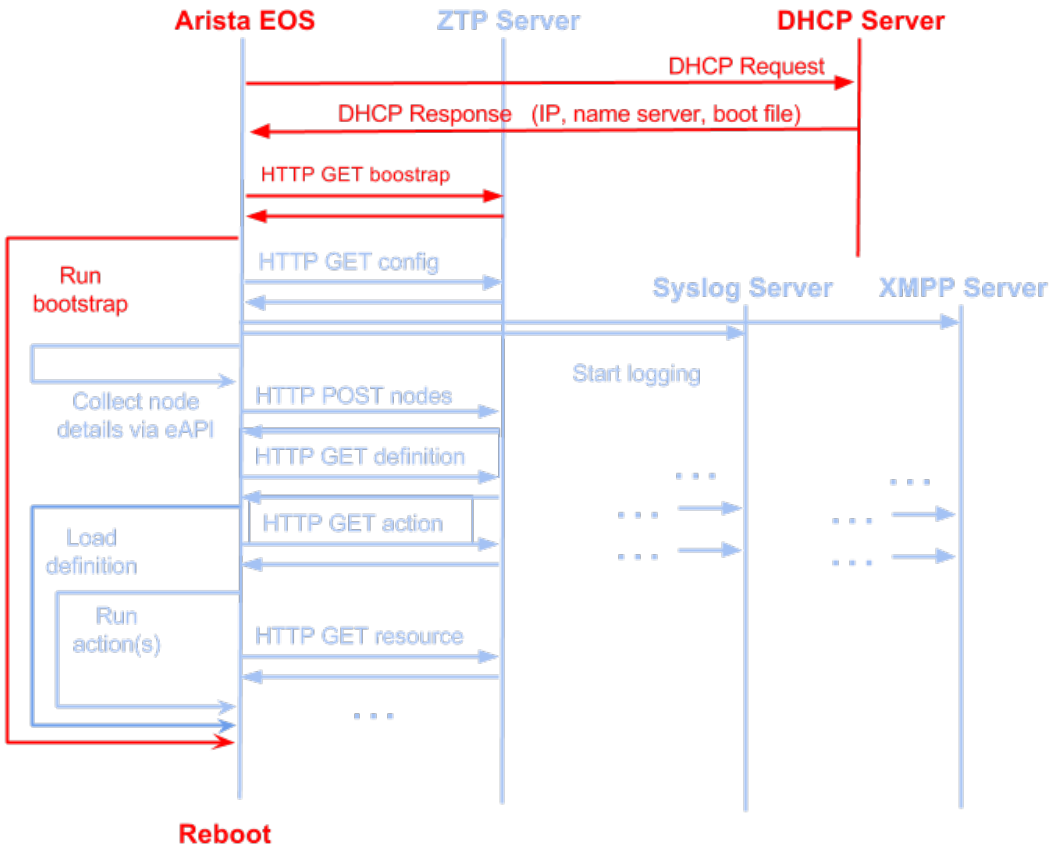
The client or **bootstrap file** is retrieved by the node via an HTTP GET request made to the ZTPServer (the URL of the file is retrieved via DHCP option 67). This file executes locally and gathers system and LLDP information from the node and sends it back to the ZTPServer. Once the ZTPServer processes the information and confirms that it can provision the node, the client makes a request to the server for a definition file - this file will contain the list of all actions which need to be executed by the node in order to provision itself.

Throughout the provisioning process the bootstrap client can log all steps via both local and remote syslogs, as well as XMPP.

ZTP Client-Server Message Flows

The following diagram show the flow of information during the bootstrap process. The lines in **red** correspond to the ZTP feature in EOS, while the lines in **blue** highlight the ZTPServer operation:

(Red indicates Arista EOS flows. Blue indicates the bootstrap client.)



Topology Validation

```
- name: standard leaf definition
  definition: dc-1/pod-1/leaf_template
  variables:
    - not_spine: excludes('spine')
    - any_spine: regex('spine\d+')
    - any_pod: includes('pod')
  interfaces:
    - Ethernet1: any_spine:Ethernet1/1
    - Ethernet2: pod1-spine2:any
    - any: excludes('spine1'):Ethernet49
    - any: excludes('spine2'):Ethernet49
    - Ethernet49:
        device: not_spine
        port: en0
    - Ethernet50:
        device: includes('spine')
         port: Ethernet50
```

ZTPServer provides a powerful topology validation engine via either `neighbordb` or `pattern` files. As part of

the bootstrap process for each node, the LLDP information received on all ports is sent to the ZTPServer and matched against either `neighbordb` or a node-specific `pattern` file (if a node is already configured on the server). Both are YAML files that are use a simple format to express strongly and loosely typed topology patterns. Pattern entries are processed top down and can include local or globally-defined variables (including regular expressions).

Patterns in `neighbordb` match nodes to definitions (dynamic mode), while node-specific pattern files are used for cabling and connectivity validation (static mode).

Topology-validation can be disabled:

- globally (`disable_topology_validation=true` in the server's global configuration file) OR
- on a per-node basis, using open patterns in the pattern files (see the *Pattern file configuration* section for more details)

Operational modes

There are several operational modes for ZTPServer, explained below. See *Neighbordb pattern examples* to see how to use them.

System ID-based provisioning with no topology validation

Via node-specific folder:

- a folder corresponding to the node's system ID is created on the server before bootstrap
- a definition file, startup-config file or both is/are placed in the folder
- topology validation is disabled globally (in the global configuration file) or via an open pattern in the pattern file located in the node-specific folder

Via neighbordb:

- a pattern which matches the node's system ID is created in neighbordb before bootstrap
- neighbordb pattern points to a definition file
- neighbordb pattern contains no topology information (LLDP neighbors)
- a node-specific folder with the definition and an open pattern will be created during the bootstrap process

System ID-based provisioning with topology validation

Via node-specific folder:

- a folder corresponding to the node's system ID is created on the server before bootstrap
- a definition file, startup-config file or both is/are placed in the folder
- topology validation is enabled globally (in the global configuration file) and the topology information is configured in the pattern file located in the node-specific folder

Via neighbordb:

- a pattern which matches the node's system ID is created in neighbordb before bootstrap
- neighbordb pattern points to a definition file
- neighbordb pattern contains topology information (LLDP neighbors)
- a node-specific folder with the definition and a pattern containing the matched toplogy information will be created during the bootstrap process

Topology-based provisioning

- a pattern which matches the topology information (LLDP neighbord) is created in neighbordb before bootstrap
- neighbordb pattern points to a definition file
- a node-specific folder with the definition and a pattern containing the matched toplogy information will be created during the bootstrap process

Installation

Requirements

Server:

- Python 2.7 or later (https://www.python.org/download/releases)
- routes 2.0 or later (https://pypi.python.org/pypi/Routes)
- webob 1.3 or later (http://webob.org/)
- PyYaml 3.0 or later (http://pyyaml.org/)

Client:

- EOS 4.12.0 or later (ZTPServer 1.1+)
- EOS 4.13.3 or later (ZTPServer 1.0)

Note: We recommend using a Linux distribution which has Python 2.7 as its standard Python install (e.g. yum in Centos requires Python 2.6 and a dual Python install can be fairly tricky and buggy). This guide was written based ZTPServer v1.1.0 installed on Fedora 20.

Installation Options

- *PyPI Package (pip install)*
- *Manual installation*

Turn-key VM Creation

The turn-key VM option leverages Packer to auto generate a VM on your local system. Packer.io automates the creation of the ZTPServer VM. All of the required packages and dependencies are installed and configured. The current Packer configuration allows you to choose between VirtualBox or VMWare as your hypervisor and each can support Fedora 20 or Ubuntu Server 12.04.

VM Specification:

- 7GB Hard Drive
- 2GB RAM
- Hostname ztps.ztps-test.com
  - eth0 (NAT) DHCP
  - eth1 (hostonly) 172.16.130.10
- Firewalld/UFW disabled
- Users
  - root/eosplus
  - ztpsadmin/eosplus
- Python 2.7.5 with PIP
- DHCP installed with Option 67 configured (eth1 only)
- BIND DNS server installed with zone ztps-test.com
  - wildcard forwarding rule passing all other queries to 8.8.8.8
  - SRV RR for im.ztps-test.com
- rsyslog-ng installed; Listening on UDP and TCP (port 514)
- ejabberd (XMPP server) configured for im.ztps-test.com
  - XMPP admin user: ztpsadmin/eosplus
- httpd installed and configured for ZTPServer (mod_wsgi)
- ZTPServer installed
- ztpserver-demo repo files pre-loaded

See the Packer VM code and documentation as well as the ZTPServer demo files for the Packer VM.

PyPI Package (pip install)

ZTPServer may be installed as a PyPI package.

This option assumes you have a server with Python and pip pre-installed. See installing pip.

Once pip is installed, type:

```
bash-3.2$ pip install ztpserver
```

The pip install process will install all dependencies and run the install script, leaving you with a ZTPServer instance ready to configure.

Manual installation

Download source:

- Latest Release on GitHub
  - Previous releases
- Active Stable: (GitHub) (ZIP) (TAR)
- Development: (GitHub) (ZIP) (TAR)

Once the above system requirements are met, you can use the following git command to pull the develop branch into a local directory on the server where you want to install ZTPServer:

```
bash-3.2$ git clone https://github.com/arista-eosplus/ztpserver.git
```

Or, you may download the zip or tar archive and expand it.

```
bash-3.2$ wget https://github.com/arista-eosplus/ztpserver/tarball/master
bash-3.2$ tar xvf <filename>
  or
bash-3.2$ unzip <filename>
```

Change in to the ztpserver directory, then checkout the release desired:

```
bash-3.2$ cd ztpserver
bash-3.2$ git checkout v1.1.0
```

Execute `setup.py` to build and then install ZTPServer:

```
[user@localhost ztpserver]$ sudo python setup.py build
running build
running build_py
...

[root@localhost ztpserver]# sudo python setup.py install
running install
running build
running build_py
running install_lib
...
```

Upgrading

Upgrading ZTP Server is based on the method of installation:

- PyPI (pip):

```
sudo pip install --upgrade ztpserver
```

- Manual, Packer-VM, GitHub installs:

```
cd ztpserver/
sudo ./utils/refresh_ztps -b <branch>
```

The ztpserver/ directory, above, should be a git repository (where the files were checked out). The `branch` identifier may be any version identifier (1.3.2, 1.1), or an actual branch on github such as `master` (released), or `develop` (development).

- RPM:

```
sudo rpm -Uvh ztpserver-<version>.rpm
```

Additional services

Note: If using the *Turn-key VM Creation*, all of the steps, below, will have been completed, please reference the VM documentation.

Allow ZTPServer Connections In Through The Firewall

Be sure your host firewall allows incoming connections to ZTPServer. The standalone server runs on port TCP/8080 by default.

Firewalld examples:

- Open TCP/<port> through firewalld `bash-3.2$ firewall-cmd --zone=public --add-port=<port>/tcp [--permanent]`
- Stop firewalld `bash-3.2$ systemctl stop firewalld`
- Disable firewalld `bash-3.2$ systemctl disable firewalld`

Note: If using the *Turn-key VM Creation*, all the steps from below will be been completed automatically.

Configure the DHCP Service

Set up your DHCP infrastructure to server the full path to the ZTPServer bootstrap file via option 67. This can be performed on any DHCP server. Below you can see how you can do that for ISC dhcpd.

Get dhcpd:

RedHat: `bash-3.2$ sudo yum install dhcp`

Ubuntu: `bash-3.2$ sudo apt-get install isc-dhcp-server`

Add a network (in this case 192.168.100.0/24) for servicing DHCP requests for ZTPServer:

```
subnet 192.168.100.0 netmask 255.255.255.0 {
  range 192.168.100.200 192.168.100.205;
  option routers 192.168.100.1;
  option domain-name-servers <ipaddr>;
  option domain-name "<org>";

  # Only return the bootfile-name to Arista devices
  class "Arista" {
    match if substring(option vendor-class-identifier, 0, 6) = "Arista";
    # Interesting bits:
    # Relay agent IP address
```

```
    # Option-82: Agent Information
    #     Suboption 1: Circuit ID
    #       Ex: 45:74:68:65:72:6e:65:74:31 ==> Ethernet1
    option bootfile-name "http://<ztp_hostname_or_ip>:<port>/bootstrap";
  }
}
```

Enable and start the dhcpd service

RedHat (and derivative Linux implementations)

```
bash-3.2# sudo /usr/bin/systemctl enable dhcpd.service  bash-3.2# sudo
/usr/bin/systemctl start dhcpd.service
```

Ubuntu (and derivative Linux implementations)

```
bash-3.2# sudo /usr/sbin/service isc-dhcp-server start
```

Check that /etc/init/isc-dhcp-server.conf is configured for automatic startup on boot.

Edit the global configuration file located at `/etc/ztpserver/ztpserver.conf` (if needed). See the *Global configuration file* options for more information.

Startup

HTTP Server Deployment Options

ZTPServer is a Python WSGI compliant application that can be deployed behind any WSGI web server or run as a standalone application.

After initial startup, any change to `ztpserver.conf` will require a server restart. However, all other files are read on-demand, therefore no server restart is required to pick up changes in definitions, neighbordb, resources, etc.

Note: The `ztps` standalone server executable is for demo and testing use ONLY. It is NOT recommended for production use!

Apache (mod_wsgi)

If using Apache, this section provides instructions for setting up ZTPServer using mod_wsgi. This section assumes the reader is familiar with Apache and has already installed mod_wsgi. For details on how to install mod_wsgi, please see the modwsgi Quick Installation Guide.

To enable ZTPServer for an Apache server, we need to add the following WSGI configuration to the Apache config. A good location might be to create `/etc/httpd/conf.d/ztpserver.conf` or /etc/apache2/sites-enabled/ztpserver.conf:

```
LoadModule wsgi_module modules/mod_wsgi.so
Listen 8080

<VirtualHost *:8080>

    WSGIDaemonProcess ztpserver user=www-data group=www-data threads=50
    WSGIScriptAlias / /etc/ztpserver/ztpserver.wsgi
    # Required for RHEL
    #WSGISocketPrefix /var/run/wsgi

    <Location />
        WSGIProcessGroup ztpserver
        WSGIApplicationGroup %{GLOBAL}

        # For Apache <= 2.2, use Order and Allow
        Order deny,allow
        Allow from all
        # For Apache >= 2.4, Allow is replaced by Require
        Require all granted
    </Location>

    # Override default logging locations for Apache
    #ErrorLog /path/to/ztpserver_error.log
    #CustomLog /path/to/ztpserver_access.log
</VirtualHost>
```

WSGIScriptAlias should point to the ztpserver.wsgi file which is installed by default under /etc/ztpserver/ztpserver.wsgi. You will notice that the `<Location />` directive is set to the root directory. This will enable ZTPServer to listen at the base server URL:

`http://<host_ip>:8080/bootstrap`

If you would like to run the ZTPServer under a subdirectory, leave the Apache configuration as it is listed above and modify the ZTPServer configuration to include the URL path prefix (`/ztpserver` in this example).

For example, edit the default configuration file found at `/etc/ztpserver/ztpserver.conf` by modifying or adding the following line under the [default] section:

`server_url = http://<host_ip>:8080/ztpserver/`

where /ztpserver/ is the subdirectory you would like the wsgi to listen. Once completed, restart Apache and you should now be able to access your ZTPServer at the specified URL. To test, simply use curl - for example:

`curl http://<host_ip>:8080/ztpserver/bootstrap`

If everything is configured properly, curl should be able to retrieve the bootstrap script. If there is a problem, all of the ZTPServer log messages should be available under the Apache server error logs. See the `ErrorLog` directive in your Apache configuration to determine the location of the error log.

Note: File Permissions - Apache mod_wsgi will run ztpserver.wsgi as the specified system user in your Apache config. This use must be able to read/write to the files in `/usr/share/ztpserver` (or whereever you created your data_root.)

Note: SELinux - Apache will need to read and write to files in `/usr/share/ztpserver`. Therefore, you might need to update/assign an SELinux user/role/type to these files. You can do something like `chcon -R -h system_u:object_r:httpd_sys_script_rw_t /usr/share/ztpserver` to accomplish that.

Standalone debug server

Note: ZTPServer ships with a single-threaded server that is sufficient for testing or demonstration, only. It is not recommended for use with more than 10 nodes.

To start the standalone ZTPServer, exec the ztps binary:

```
[root@ztpserver ztpserver]# ztps
INFO: [app:115] Logging started for ztpserver
INFO: [app:116] Using repository /usr/share/ztpserver
Starting server on http://<ip_address>:<port>
```

The following options may be specified when starting the ztps binary:

```
-h, --help            show this help message and exit
--version, -v         Displays the version information
--conf CONF, -c CONF  Specifies the configuration file to use
--validate-config, -V
                      Validates config files
--debug               Enables debug output to the STDOUT
--clear-resources, -r
                      Clears all resource files
```

Assuming that the DHCP server is serving DHCP offers which include the path to the ZTPServer bootstrap script in Option 67 and that the EOS nodes can access the bootstrap file over the network, the provisioning process should now be able to automatically start for all the nodes with no startup configuration.

Configuration

Overview

The ZTPServer uses a series of YAML files to provide its various configuration and databases. Use of the YAML format makes the files easier to read and makes it easier and more intuitive to add/update entries (as opposed to other files formats such as JSON, or binary formats such as SQL).

The ZTPServer components are housed in a single directory defined by the `data_root` variable in the global configuration file. The directory location will vary depending on the configuration in `/etc/ztpserver/ztperserver.conf`.

The following directory structure is normally used:

```
[data_root]
    bootstrap/
        bootstrap
        bootstrap.conf
    nodes/
        <unique_id)>/
            startup-config
            definition
            pattern
            config-handler
            .node
            attributes
    actions/
    files/
    definitions/
    resources/
    neighbordb
```

All configuration files can be validated using:

```
(bash)# ztps --validate
```

Global configuration file

The global ZTPServer configuration file can be found at `/etc/ztpserver/ztpserver.conf`. It uses the INI format (for details, see top section of Python configparser).

An alternative location for the global configuration file may be specified by using the `--conf` command line option:

e.g.

```
(bash)# ztps --help
usage: ztpserver [options]

optional arguments:
  -h, --help            show this help message and exit
  --version, -v         Displays the version information
  **--conf CONF, -c CONF  Specifies the configuration file to use**
  --validate-config, -V
                        Validates config files
  --debug               Enables debug output to the STDOUT
  --clear-resources, -r
                        Clears all resource files
(bash)# ztps --conf /var/ztps.conf
```

If the global configuration file is updated, the server must be restarted in order to pick up the new configuration.

```
[default]

# Location of all ztps boostrap process data files
# default= /usr/share/ztpserver
data_root=<PATH>

# UID used in the /nodes structure
# default=serialnum
identifier=<serialnum | systemmac>

# Server URL to-be-advertised to clients (via POST replies) during the bootstrap␣
↪process
# default=http://ztpserver:8080
server_url=<URL>

# Enable local logging
# default=True
logging=<True | False>

# Enable console logging
# default=True
console_logging=<True | False>

# Console logging format
# default=%(asctime)-15s:%(levelname)s:[%(module)s:%(lineno)d] %(message)s
console_logging_format=<(Python)logging format>

# Globally disable topology validation in the bootstrap process
# default=False
disable_topology_validation=<True | False>

[server]
# Note: this section only applies to using the standalone server.  If
# running under a WSGI server, these values are ignored

# Interface to which the server will bind to (0:0:0:0 will bind to
# all available IPv4 addresses on the local machine)
# default=0.0.0.0
interface=<IP addr>

# TCP listening port
# default=8080
```

```
port=<TCP port>

[bootstrap]
# Bootstrap filename (file located in <data_root>/bootstrap)
# default=bootstrap
filename=<name>

[neighbordb]
# Neighbordb filename (file located in <data_root>)
# default=neighbordb
filename=<name>
```

Note: Configuration values may be overridden by setting environment variables, if the configuration attribute supports it. This is mainly used for testing and should not be used in production deployments.

Configuration values that support environment overrides use the `environ` keyword, as shown below:

```
runtime.add_attribute(StrAttr(
    name='data_root',
    default='/usr/share/ztpserver',
    environ='ZTPS_DEFAULT_DATAROOT'
))
```

In the above example, the `data_root` value is normally configured in the [default] section as `data_root`; however, if the environment variable `ZTPS_DEFAULT_DATAROOT` is defined, it will take precedence.

Bootstrap configuration

`[data_root]/bootstrap/` contains files that control the bootstrap process of a node.

- **bootstrap** is the base bootstrap script which is going to be served to all clients in order to control the bootstrap process. Before serving the script to the clients, the server replaces any references to $SERVER with the value of `server_url` in the global configuration file.
- **bootstrap.conf** is a configuration file which defines the local logging configuration on the nodes (during the bootstrap process). The file is loaded on on-demand.

  e.g.

```
---
logging:
  -
    destination: "ztps.ztps-test.com:514"
    level: DEBUG
  - destination: file:/tmp/ztps-log
    level: DEBUG
  - destination: ztps-server:1234
    level: CRITICAL
  - destination: 10.0.1.1:9000
    level: CRITICAL
xmpp:
  domain: im.ztps-test.com
  username: bootstrap
  password: eosplus
  rooms:
```

```
    - ztps
    - ztps-room2
```

Note: In order for XMPP logging to work, a non-EOS user need to be connected to the room specified in bootstrap.conf, before the ZTP process starts. The room has to be created (by the non-EOS user) before the bootstrap client starts logging the ZTP process via XMPP.

Static provisioning - overview

A node can be statically configured on the server as follows:

- create a new directory under `[data_root]/nodes`, using the system's unique_id as the name
- create/symlink a *startup-config* or *definition* file in the newly-created folder
- if topology validation is enabled, also create/symlink a *pattern* file
- optionally, create *config-handler* script which is run whenever a PUT startup-config request succeeds

Static provisioning - startup_config

`startup-config` provides a static startup-configuration for the node. If this file is present in a node's folder, when the node sends a GET request to `/nodes/<unique_id>`, the server will respond with a static definition that includes:

- a **replace_config** action which will install the configuration file on the switch (see *actions* section below for more on this). This action will be placed **first** in the definition.
- all the **actions** from the local **definition** file (see definition section below for more on this) which have the `always_execute` attribute set to `True`

Static provisioning - definition

The **definition** file contains the set of actions which are going to be performed during the bootstrap process for a node. The definition file can be either: **manually created** OR **auto-generated by the server** when the node matches one of the patterns in **neighbordb** (in this case the definition file is generated based on the definition file associated with the matching pattern in **neighbordb**).

```
name: <system name>

actions:
  -
    action: <action name>

    attributes:                     # attributes at action scope
        always_execute: True        # optional, default False
        <key>: <value>
        <key>: <value>

    onstart:   <msg>                # message to log before action is executed
    onsuccess: <msg>                # message to log if action execution succeeds
    onfailure: <msg>                # message to log if action execution fails
  ...
```

```
attributes:                          # attributes at global scope
    <key>: <value>
    <key>: <value>
    <key>: <value>
```

Static provisioning - attributes

Attributes are either key/value pairs, key/dictionary pairs, key/list pairs or key/reference pairs. They are all sent to the client in order to be passed in as arguments to actions.

Here are a few examples:

- key/value:

```
attributes:
    my_attribute : my_value
```

- key/dictionary

```
attributes:
    my_dict_attribute:
        key1: value1
        key2: value2
```

- key/list:

```
attributes:
  list_name:
    - my_value1
    - my_value2
    - my_valueN
```

- key/reference:

```
attributes:
    my_attribute : $my_other_attribute
```

key/reference attributes are identified by the fact that the value starts with the '$' sign, followed by the name of another attribute. They are evaluated before being sent to the client.

Example:

```
attributes:
    my_other_attribute: dummy
    my_attribute : $my_other_attribute
```

will be evaluated to:

```
attributes:
    my_other_attribute: dummy
    my_attribute : dummy
```

If a reference points to a non-existing attribute, then the variable substitution will result in a value of *None*.

Note: Only **one level of indirection** is allowed - if multiple levels of indirection are used, then the data sent to the client will contain unevaluated key/reference pairs in the attributes list (which might lead to failures or unexpected results in the client).

The values of the attributes can be either strings, numbers, lists, dictionaries, or references to other attributes or plugin references for allocating resources.

Plugins can be used to allocate resources on the server side and then pass the result of the allocation back to the client via the definition. The supported plugins are:

- **allocate(resource_pool)** - allocates an available resource from a file-based resource pool
- **sqlite(resource_pool)** - allocates an available resource from a sqlite database

Note: Plugins can only be referenced with strings as arguments, currently. See section on *add_config* action for examples.

Attributes can be defined in three places:

- in the definition, at action scope
- in the definition, at global scope
- in the node's attributes file (see below)

`attributes` is a file which can be used in order to store attributes associated with the node's definition. This is especially useful whenever multiple nodes share the same definition - in that case, instead of having to edit each node's definition in order to add the attributes (at the global or action scope), all nodes can share the same definition (which might be symlinked to their individual node folder) and the user only has to create the attributes file for each node. The `attributes` file should be a valid key/value YAML file.

```
<key>: <value>
<key>: <value>
...
```

For key/value, key/list and and key/reference attributes, in case of conflicts between the three scopes, the following order of precidence rules are applied to determine the final value to send to the client:

1. action scope in the definition takes precedence
2. attributes file comes next
3. global scope in the definition comes last

For key/dict attributes, in case of conflicts between the scopes, the dictionaries are merged. In the event of dictionary key conflicts, the same precidence rules from above apply.

Static provisioning - pattern

The `pattern` file a way to validate the node's topology during the bootstrap process (if topology validation is enabled). The pattern file can be either:

- manually created
- auto-generated by the server, when the node matches one of the patterns in `neighbordb` (the pattern that is matched in `neighbordb` is, then, written to this file and used for topology validation in subsequent re-runs of the bootstrap process)

The format of a pattern is very similar to the format of `neighordb` (see *neighbordb* section below):

```
variables:
    <variable_name>: <function>
...

name: <single line description of pattern>                # optional
interfaces:
    - <port_name>:<system_name>:<neighbor_port_name>
    - <port_name>:
        device: <system_name>
        port: <neighbor_port_name>
...
```

If the pattern file is missing when the node makes a GET request for its definition, the server will log a message and return either:

- 400 (BAD_REQUEST) if topology validation is enabled
- 200 (OK) if topology validation is disabled

If topology validation is enabled globally, the following patterns can be used in order to disable it for a particular node:

- match **any** node which has at least one LLDP-capable neighbor:

```
name: <pattern name>
interfaces:
    - any: any:any
```

OR

- match **any** node which has no LLDP-capable neighbors:

```
name: <pattern name>
interfaces:
    - none: none:none
```

Static provisioning - config-handler

The `config-handler` file can be any script which can be executed on the server. The script will be executed every time a PUT startup-config request succeeds for the node.

The script can be used for raising alarms, performing checks, submitting the startup-config file to a revision control system, etc.

Static provisioning - log

The `.node` file contains a cached copy of the node's details that were received during the POST request the node makes to `/nodes (URI)`. This cache is used to validate the node's neighbors against the `pattern` file, if topology validation is enabled (during the GET request the node makes in order to retrieve its definition).

The `.node` is created automatically by the server and should not be edited manually.

Example .node file:

```
{"neighbors": {"Management1": [{"device": "ztps.ztps-test.com",
                                "port": "0050.569b.9ba5"}
                               ],
```

```
                "Ethernet2": [{"device": "veos-dc1-pod1-spine1",
                               "port": "0050.569a.9321"}
                              ]
              },
 "model": "vEOS",
 "version": "4.13.7M",
 "systemmac": "005056b863ac"
}
```

Dynamic provisioning - overview

A node can be dynamically provisioned by creating a matching `neighbordb` (`[data_root]/neighbordb`) entry which maps to a definition. The entry can potentially match multiple nodes. The associated definition should be created in [data_root]/definitions/.

Dynamic provisioning - neighbordb

The `neighbordb` YAML file defines mappings between patterns and definitions. If a node is not already configured via a static entry, then the node's topology details are attempted to be matched against the patterns in `neighbordb`. If a match is successful, then a node definition will be automatically generated for the node (based on the mapping in neighbordb).

There are 2 types of patterns supported in neighbordb: node-specific (containing the **node** attribute, which refers to the unique_id of the node) and global patterns.

Rules:

- if multiple node-specific entries reference the same unique_id, only the first will be in effect - all others will be ignored
- if both the **node** and **interfaces** attributes are specified and a node's unique_id is a match, but the topology information is not, then the overall match will fail and the global patterns will not be considered
- if there is no matching node-specific pattern for a node's unique_id, then the server will attempt to match the node against the global patterns (in the order they are specified in `neighbordb`)
- if a node-specific pattern matches, the server will automatically generate an open pattern in the node's folder. This pattern will match any device with at least one LLDP-capable neighbor. Example: `any:  any:any`

```
---
variables:
    variable_name: function
...
patterns:
    - name: <single line description of pattern>
      definition: <defintion_url>
      node: <unique_id>
      config-handler: <config-handler>
      variables:
        <variable_name>: <function>
      interfaces:
        - <port_name>: <system_name>:<neighbor_port_name>
        - <port_name>:
            device: <system_name>
            port: <neighbor_port_name>
...
```

Note: Mandatory attributes: **name**, **definition**, and either **node**, **interfaces** or both.

Optional attributes: **variables**, **config-handler**.

variables

The variables can be used to match the remote device and/or port name (`<system_name>`, `<neighbor_port_name>` above) for a neighbor. The supported values are:

string same as exact(string) from below

exact (pattern) defines a pattern that must be matched exactly (Note: this is the default function if another function is not specified)

regex (pattern) defines a regex pattern to match the node name against

includes (string) defines a string that must be present in system/port name

excludes (string) defines a string that must not be present in system/port name

node: unique_id

Serial number or MAC address, depending on the global 'identifier' attribute in **ztpserver.conf**.

interfaces: port_name

Local interface name - supported values:

- **Any interface**
  - any
- **No interface**
  - none
- **Explicit interface**
  - Ethernet1
  - Ethernet2/4
  - Management1
- **Interface list/range**
  - Ethernet1-2
  - Ethernet1,3
  - Ethernet1-2,3/4
  - Ethernet1-2,4
  - Ethernet1-2,4,6
  - Ethernet1-2,4,6,8-9
  - Ethernet4,6,8-9
  - Ethernet10-20

  - Ethernet1/3-2/4 *
  - Ethernet3-$ *
  - Ethernet1/10-$ *
- **All Interfaces on a Module**
  - Ethernet1/$ *

Note: * Planned for future releases.

system_name:neighbor_port_name

Remote system and interface name - supported values (STRING = any string which does not contain any white spaces):

- `any`: interface is connected
- `none`: interface is NOT connected
- `<STRING>:<STRING>`: interface is connected to specific device/interface
- `<STRING>` (Note: if only the device is configured, then 'any' is implied for the interface. This is equal to `<DEVICE>:any`): interface is connected to device
- `<DEVICE>:any`: interface is connected to device
- `<DEVICE>:none`: interface is NOT connected to device (might be connected or not to some other device)
- `$<VARIABLE>:<STRING>`: interface is connected to specific device/interface
- `<STRING>:<$VARIABLE>`: interface is connected to specific device/interface
- `$<VARIABLE>:<$VARIABLE>`: interface is connected to specific device/interface
- `$<VARIABLE>` ('any' is implied for the interface. This is equal to `$<VARIABLE>:any`): interface is connected to device
- `$<VARIABLE>:any`: interface is connected to device
- `$<VARIABLE>:none`: interface is NOT connected to device (might be connected or not to some other device)

port_name: system_name:neighbor_port_name

Negative constraints

1. `any: DEVICE:none`: no port is connected to DEVICE
2. `none: DEVICE:any`: same as above
3. `none: DEVICE:none`: same as above
4. `none: any:PORT`: no device is connected to PORT on any device
5. `none: DEVICE:PORT`: no device is connected to DEVICE:PORT
6. `INTERFACES: any:none`: interfaces not connected
7. `INTERFACES: none:any`: same as above
8. `INTERFACES: none:none`: same as above
9. `INTERFACES: none:PORT`: interfaces not connected to PORT on any device

10. `INTERFACES: DEVICE:none`: interfaces not connected to DEVICE
11. `any: any:none`: bogus, will prevent pattern from matching anything
12. `any: none:none`: bogus, will prevent pattern from matching anything
13. `any: none:any`: bogus, will prevent pattern from matching anything
14. `any: none:PORT`: bogus, will prevent pattern from matching anything
15. `none: any:any`: bogus, will prevent pattern from matching anything
16. `none: any:none`: bogus, will prevent pattern from matching anything
17. `none: none:any`: bogus, will prevent pattern from matching anything
18. `none: none:none`: bogus, will prevent pattern from matching anything
19. `none: none:PORT`: bogus, will prevent pattern from matching anything

Positive constraints

1. `any: any:any`: "Open pattern" matches anything
2. `any: any:PORT`: matches any interface connected to any device's PORT
3. `any: DEVICE:any`: matches any interface connected to DEVICE
4. `any: DEVICE:PORT`: matches any interface connected to DEVICE:PORT
5. `INTERFACES: any:any`: matches if local interfaces is one of INTERFACES
6. `INTERFACES: any:PORT`: matches if one of INTERFACES is connected to any device's PORT
7. `INTERFACES: DEVICE:any`: matches if one of INTERFACES is connected to DEVICE
8. `INTERFACES: DEVICE:PORT`: matches if one of INTERFACES is connected to DEVICE:PORT

Definitions

`[data_root]/definitions/` contains a set of shared definition files which can be associated with patterns in `neighbordb` (see the *Dynamic provisioning - neighbordb* section below) or added to/symlink-ed from nodes' folders.

See *Static provisioning - definition* for more.

Actions

`[data_root]/actions/` contains the set of all actions available for use in definitions.

New custom actions to-be referenced from definitions can be added to `[data_root]/actions/`. These will be loaded on-demand and do not require a restart of the ZTPServer. See `[data_root]/actions` for examples.

Action	Description	Required Attributes
add_config	Adds a block of configuration to the final startup-config file	url
copy_file	Copies a file from the server to the destination node	src_url, dst_url, overwrite, mode
install_cli_plugin	Installs a new EOS CLI plugin and configures rc.eos	url
install_extension	Installs a new EOS extension	extension_url, autoload, force
install_image	Validates and installs a specific version of EOS	url, version
replace_config	Sends an entire startup-config to the node (overrides (overrides add_config)	url
send_email	Sends an email to a set of recipients routed through a relay host. Can include file attachments	smarthost, sender, receivers, subject, body, attachments, commands
run_bash_script	Run bash script during bootstrap.	url
run_cli_commands	Runs CLI commands during bootstrap.	url

Additional details on each action are available in the *Actions* module docs.

e.g.

Assume that we have a block of configuration that adds a list of NTP servers to the startup configuration. The action would be constructed as such:

```
actions:
    - name: configure NTP
      action: add_config
      attributes:
        url: /files/templates/ntp.template
```

The above action would reference the `ntp.template` file which would contain configuration commands to configure NTP. The template file could look like the following:

```
ntp server 0.north-america.pool.ntp.org
ntp server 1.north-america.pool.ntp.org
ntp server 2.north-america.pool.ntp.org
ntp server 3.north-america.pool.ntp.org
```

When this action is called, the configuration snippet above will be appended to the `startup-config` file.

The configuration templates can also contains **variables**, which are automatically substituted during the action's execution. A variable is marked in the template via the '$' symbol.

e.g. Let's assume a need for a more generalized template that only needs node specific values changed (such as a hostname and management IP address). In this case, we'll build an action that allows for **variable substitution** as follows.

```
actions:
    - name: configure system
      action: add_config
      attributes:
        url: /files/templates/system.template
        variables:
            hostname: veos01
            ipaddress: 192.168.1.16/24
```

The corresponding template file `system.template` will provide the configuration block:

```
hostname $hostname
!
interface Management1
    description OOB interface
```

```
    ip address $ipaddress
    no shutdown
```

This will result in the following configuration being added to the `startup-config`:

```
hostname veos01
!
interface Management1
    description OOB interface
    ip address 192.168.1.16/24
    no shutdown
```

Note that in each of the examples, above, the template files are just standard EOS configuration blocks.

Plugins for allocating resources

Plugins for allocating resources from resource pools are located in `[data_root]/plugins/` and are referenced by `<filename>(<resource_pool>)`.

Each plugin contains a `main` function with the following signature:

def main(node_id, pool): ...

where:

- `node_id` is the unique_id of the node being provisioned
- `pool` is the name of the resource pool from which an attribute is being allocated

New custom plugins to-be referenced from definitions can be added to `[data_root]/plugins/`. These will be loaded on-demand and do not require a restart of the ZTPServer. See `[data_root]/plugins/test` for a very basic example.

allocate(resource_pool)

`[data_root]/resources/` contains global resource pools from which attributes in definitions can be allocated.

The resource pools provide a way to dynamically allocate a resource to a node when the node definition is created. The resource pools are key/value YAML files that contain a set of resources to be allocated to a node.

```
<value1>: <"null"|node_identifier>
<value2>: <"null"|node_identifier>
```

In the example below, a resource pool contains a series of 8 IP addresses to be allocated. Entries which are not yet allocated to a node are marked using the `null` descriptor.

```
192.168.1.1/24: null
192.168.1.2/24: null
192.168.1.3/24: null
192.168.1.4/24: null
192.168.1.5/24: null
192.168.1.6/24: null
192.168.1.7/24: null
192.168.1.8/24: null
```

When a resource is allocated to a node's definition, the first available null value will be replaced by the node's unique_id. Here is an example:

```
192.168.1.1/24: 001c731a2b3c
192.168.1.2/24: null
192.168.1.3/24: null
192.168.1.4/24: null
192.168.1.5/24: null
192.168.1.6/24: null
192.168.1.7/24: null
192.168.1.8/24: null
```

On subsequent attempts to allocate the resource to the same node, ZTPS will first check to see whether the node has already been allocated a resource from the pool. If it has, it will reuse the resource instead of allocating a new one.

In order to free a resource from a pool, simply turn the value associated to it back to `null`, by editing the resource file.

Alternatively, `$ztps --clear-resources` can be used in order to free all resources in all file-based resource files.

sqlite(resource_pool)

Allocates a resource from a pre-filled sqlite database. The database is defined by the global variable, 'DB_URL' within the plugin. The database can include multiple tables, but the value passed into the 'sqlite(resource_pool)' function will be used to look for an available resource.

Table structure should be as follows with the exact column names:

node_id	key
NULL	1.1.1.1
NULL	1.1.1.2
NULL	1.1.1.3

Which can be created with statements like:

```
CREATE TABLE `mgmt_subnet`(key TEXT, node_id TEXT)
```

and add entries with:

```
INSERT INTO `mgmt_subnet` VALUES('1.1.1.1', NULL)
```

When a resource is added, the node_id row will be updated to include the System ID from the switch.

node_id	key
001122334455	1.1.1.1
NULL	1.1.1.2
NULL	1.1.1.3

On subsequent attempts to allocate the resource to the same node, ztpserver will first check to see whether the node has already been allocated a resource from the pool. If it has, it will reuse the resource instead of allocating a new one.

Definition example:

```
actions:
  -
    action: add_config
    attributes:
      url: files/templates/ma1.templates
      variables:
        ipaddress: sqlite('mgmt_subnet')
    name: "configure ma1"
```

Tip: Check out create_db.py for an example script to create a sqlite database.

Config-handlers

`[data_root]/config-handlers/` contains config-handlers which can be associated with nodes via *neighbordb*. A config-handler script is executed every time a PUT startup-config request succeeds for a node which is associated to it.

Other files

`[data_root]/files/` contains the files that actions might request from the server. For example, `[data_root]/files/images/` could contain all EOS SWI files.

Examples

Global configuration file

```
[default]
# Location of all ztps boostrap process data files
data_root = /usr/share/ztpserver

# UID used in the /nodes structure (serialnumber or systemmac)
identifier = serialnumber

# Server URL to-be-advertised to clients (via POST replies) during the bootstrap␣
↪process
server_url = http://172.16.130.10:8080
```

```
# Enable local logging
logging = True

# Enable console logging
console_logging = True

# Console logging format
console_logging_format = %(asctime)s:%(levelname)s:[%(module)s:%(lineno)d] %(message)s


# Globally disable topology validation in the bootstrap process
disable_topology_validation = False

[server]
# Note: this section only applies to using the standalone server.  If
# running under a WSGI server, these values are ignored

# Interface to which the server will bind to (0:0:0:0 will bind to
# all available IPv4 addresses on the local machine)
interface = 172.16.130.10

# TCP listening port
port = 8080

[bootstrap]
# Bootstrap filename (file located in <data_root>/bootstrap)
filename = bootstrap

[neighbordb]
# Neighbordb filename (file located in <data_root>)
filename = neighbordb
```

Dynamic neighbordb or pattern file

```
---
patterns:
#dynamic sample
  - name: dynamic_sample
    definition: tor1
    interfaces:
      - Ethernet1: spine1:Ethernet1
      - Ethernet2: spine2:Ethernet1
      - any: ztpserver:any

  - name: dynamic_sample2
    definition: tor2
    interfaces:
      - Ethernet1: spine1:Ethernet2
      - Ethernet2: spine2:Ethernet2
      - any: ztpserver:any
```

Static neighbordb and /node/<unique-id>/pattern file

```
---
patterns:
#static sample
  - name: static_node
    node: 000c29f3a39g
    interfaces:
      - any: any:any
```

Sample dynamic definition file

```
---
actions:
  -
    action: install_image
    always_execute: true
    attributes:
      url: files/images/vEOS.swi
      version: 4.13.5F
    name: "validate image"
  -
    action: add_config
    attributes:
      url: files/templates/ma1.template
      variables:
        ipaddress: allocate('mgmt_subnet')
    name: "configure ma1"
  -
    action: add_config
    attributes:
      url: files/templates/system.template
      variables:
        hostname: allocate('tor_hostnames')
    name: "configure global system"
  -
    action: add_config
    attributes:
      url: files/templates/login.template
    name: "configure auth"
  -
    action: add_config
    attributes:
      url: files/templates/ztpprep.template
    name: "configure ztpprep alias"
  -
    action: add_config
    attributes:
      url: files/templates/snmp.template
      variables: $variables
    name: "configure snmpserver"
  -
    action: add_config
    attributes:
      url: files/templates/configpush.template
      variables: $variables
```

```
    name: "configure config push to server"
  -
    action: copy_file
    always_execute: true
    attributes:
      dst_url: /mnt/flash/
      mode: 777
      overwrite: if-missing
      src_url: files/automate/ztpprep
    name: "automate reload"
attributes:
  variables:
    ztpserver: 172.16.130.10
name: tora
```

Sample templates

```
#login.template
#:::::::::::::::
username admin priv 15 secret admin
```

```
#ma1.template
#:::::::::::::::
interface Management1
  ip address $ipaddress
  no shutdown
```

```
#hostname.template
#:::::::::::::::
hostname $hostname
```

Sample resources

```
#mgmt_subnet
#:::::::::::::::
192.168.100.210/24: null
192.168.100.211/24: null
192.168.100.212/24: null
192.168.100.213/24: null
192.168.100.214/24: null
```

```
#tor_hostnames
#:::::::::::::::
veos-dc1-pod1-tor1: null
veos-dc1-pod1-tor2: null
veos-dc1-pod1-tor3: null
veos-dc1-pod1-tor4: null
veos-dc1-pod1-tor5: null
```

Neighbordb pattern examples

Example #1

```
---
- name: standard leaf definition
  definition: leaf_template
  node: ABC12345678
  interfaces:
    - Ethernet49: pod1-spine1:Ethernet1/1
    - Ethernet50:
        device: pod1-spine2
        port: Ethernet1/1
```

In example #1, the topology map would only apply to a node with system ID equal to **ABC12345678**. The following interface map rules apply:

- Interface Ethernet49 must be connected to node pod1-spine1 on port Ethernet1/1
- Interface Ethernet50 must be connected to node pod1-spine2 on port Ethernet1/1

Example #2

```
---
- name: standard leaf definition
  definition: leaf_template
  node: 001c73aabbcc
  interfaces:
    - any: regex('pod\d+-spine\d+'):Ethernet1/$
    - any:
        device: regex('pod\d+-spine1')
        port: Ethernet2/3
```

In this example, the topology map would only apply to the node with system ID equal to **001c73aabbcc**. The following interface map rules apply:

- At least one interface interface must be connected to node that matches the regular expression 'pod+-spine+' on port Ethernet1/$ (any port on module 1)
- At least one interface and not the interface which matched in the previous step must be connected to a node that matches the regular expression 'pod+-spine1' on port Ethernet2/3

Example #3

```
---
- name: standard leaf definition
  definition: dc-1/pod-1/leaf_template
  variables:
    - not_spine: excludes('spine')
    - any_spine: regex('spine\d+')
    - any_pod: includes('pod')
    - any_pod_spine: any_spine and any_pod*
  interfaces:
    - Ethernet1: $any_spine:Ethernet1/$
    - Ethernet2: $pod1-spine2:any
    - any: excludes('spine1'):Ethernet49
    - any: excludes('spine2'):Ethernet49
    - Ethernet49:
```

```
        device: $not_spine
        port: Ethernet49
    - Ethernet50:
        device: excludes('spine')
        port: Ethernet50
```

Note: * In a future release.

This example pattern could apply to any node that matches the interface map. In includes the use of variables for cleaner implementation and pattern re-use.

- Variable not_spine matches any node name where 'spine' doesn't appear in the string
- Variable any_spine matches any node name where the regular expression 'spine+' matches the name
- Variable any_pod matches any node name where that includes the name 'pod' in it
- **Variable any_pod_spine combines variables any_spine and any_pod into a complex variable that includes any name that matches the regular express 'spine+' and the name includes 'pod' (not yet supported)**
- Interface Ethernet1 must be connected to a node that matches the any_spine pattern and is connected on Ethernet1/$ (any port on module 1)
- Interface Ethernet2 must be connected to node 'pod1-spine2' on any Ethernet port
- Interface any must be connected to any node that doesn't have 'spine1' in the name and is connected on Ethernet49
- Interface any must be connected to any node that doesn't have 'spine2' in the name and wasn't already used and is connected to Ethernet49
- Interface Ethernet49 matches if it is connected to any node that matches the not_spine pattern and is connected on port 49
- Interface Ethernet50 matches if the node is connected to port Ethernet50 on any node whose name does not contain 'spine'

Example #4

```
---
- name: sample mlag definition
  definition: mlag_leaf_template
  variables:
    any_spine: includes('spine')
    not_spine: excludes('spine')
  interfaces:
    - Ethernet1: $any_spine:Ethernet1/$
    - Ethernet2: $any_spine:any
- Ethernet3: none
- Ethernet4: any
- Ethernet5:
    device: includes('oob')
    port: any
- Ethernet49: $not_spine:Ethernet49
- Ethernet50: $not_spine:Ethernet50
```

This is a similar example to #3 that demonstrates how an MLAG pattern might work.

- Variable any_spine defines a pattern that includes the word 'spine' in the name
- Variable not_spine defines a pattern that matches the inverse of any_spine

- Interface Ethernet1 matches if it is connected to any_spine on port Ethernet1/$ (any port on module 1)
- Interface Ethernet2 matches if it is connected to any_spine on any port
- Interface 3 matches so long as there is nothing attached to it
- Interface 4 matches so long as something is attached to it
- Interface 5 matches if the node contains 'oob' in the name and is connected on any port
- Interface49 matches if it is connected to any device that doesn't have 'spine' in the name and is connected on Ethernet50
- Interface50 matches if it is connected to any device that doesn't have 'spine' in the name and is connected on port Ethernet50

More examples

Additional ZTPServer file examples are available on GitHub at the ZTPServer Demo.

ZTPServer Cookbook

Installation

Recipes

Install ZTPServer from Github Source

Objective

I want to install ZTPServer from source.

Solution

To install the latest code in development:

```
# Change to desired download directory
mkdir -p ~/arista
cd ~/arista
git clone https://github.com/arista-eosplus/ztpserver.git
cd ztpserver
python setup.py build
python setup.py install
```

Or, to install a specific tagged release:

```
# Change to desired download directory
mkdir -p ~/arista
cd ~/arista
git clone https://github.com/arista-eosplus/ztpserver.git
cd ztpserver
git checkout v1.2.0
python setup.py build
python setup.py install
```

Explanation

Github is used to store the source code for the ZTPServer and the `develop` branch always contains the latest publicly available code. The first method above clones the git repo and automatically checks out the `develop` branch. We then `build` and `install` using Python.

The second method uses the `git checkout` command to set your working directory to a specific release of the ZTPServer. Both methods of installation will produce the files below.

Important Installation Files

- ZTPServer Global Configuration File: `/etc/ztpserver/ztpserver.conf`
- ZTPServer WSGI App: `/etc/ztpserver/ztpserver.wsgi`
- ZTPServer Provisioning Files: `/usr/share/ztpserver/` known as `data_root`
- Bootstrap Config File: `/usr/share/ztpserver/bootstrap/bootstrap.conf`
- Bootstrap Python Script: `/usr/share/ztpserver/bootstrap/bootstrap`

Install ZTPServer using PIP

Objective

Install ZTPServer using PyPI(pip)

Solution

This option assumes you have a server with Python and pip pre-installed. See installing pip.

Once pip is installed, type:

```
pip install ztpserver
```

Explanation

The pip install process will install all dependencies and run the install script, leaving you with a ZTPServer instance ready to configure.

Important Installation Files

- ZTPServer Global Configuration File: `/etc/ztpserver/ztpserver.conf`
- ZTPServer WSGI App: `/etc/ztpserver/ztpserver.wsgi`

- ZTPServer Provisioning Files: `/usr/share/ztpserver/` known as `data_root`
- Bootstrap Config File: `/usr/share/ztpserver/bootstrap/bootstrap.conf`
- Bootstrap Python Script: `/usr/share/ztpserver/bootstrap/bootstrap`

Client-Side Logging

Configure Syslog Logging

Objective

I want to send client logs to a syslog server or a local file during provisioning.

Solution

```
# Edit the bootstrap configuration file
admin@ztpserver:~# vi /usr/share/ztpserver/bootstrap/bootstrap.conf
```

Add any syslog servers or files, be sure to choose the level of logging:

```
---
logging:
  -
    destination: <SYSLOG-URL>:<PORT>
    level: DEBUG
  -
    destination: file:/tmp/ztps-log
    level: INFO
```

Explanation

The node will request the contents of the `bootstrap.conf` when it performs `GET /bootstrap/config`. Once the node retrieves this information it will send logs to the `destination(s):` listed under `logging:`.

Configure XMPP Logging

Objective

I want to send client logs to specific XMPP server rooms.

Solution

```
# Edit the bootstrap configuration file
admin@ztpserver:~# vi /usr/share/ztpserver/bootstrap/bootstrap.conf
```

Add any XMPP servers and associated rooms:

```
---
xmpp:
  domain: <XMPP-SERVER-URL>
  username: bootstrap
  password: eosplus
  rooms:
    - ztps
    - devops
    - admins
```

Explanation

The node will request the contents of the `bootstrap.conf` when it performs `GET /bootstrap/config` file and try to join the rooms listed with the credentials provided. Typically when joining a room, you would use a string like, `ztps@conference.xmpp-server.example.com`. Be sure to just provide the `domain: xmpp-server.example.com` leaving out the `conference` prefix.

Note: In order for XMPP logging to work, a non-EOS user need to be connected to the room specified in bootstrap.conf, before the ZTP process starts. The room has to be created (by the non-EOS user) before the bootstrap client starts logging the ZTP process via XMPP.

Server-Side Logging

Standalone - Redirect Output to file

Objective

When running the ZTPServer in Standalone Mode, the logs just fill up my console so I'd like to be able to redirect the output to a file.

Solution

With INFO level logging:

```
admin@ztpserver:~# ztps >~/ztps-console.log 2>&1 &
```

With DEBUG level logging:

```
admin@ztpserver:~# ztps --debug >~/ztps-console.log 2>&1 &
```

Explanation

Here we invoke the ztps process as usual, however we redirect the stdout messages to a predefined file. Of course, be sure that you have permission to write to the file you have listed.

Apache - View Standard Logs

Objective

I'm running the ZTPServer as a WSGI under Apache, so where do the logs go?

Solution

Typically, you can see each transaction in:

```
# Ubuntu
admin@ztpserver:~# more /var/log/apache2/access.log

# Fedora
admin@ztpserver:~# more /var/log/httpd/access_log
```

And the ZTPServer logs will be in:

```
# Ubuntu
admin@ztpserver:~# more /var/log/apache2/error.log

# Fedora
admin@ztpserver:~# more /var/log/httpd/error_log
```

Explanation

These locations are the default on most standard Apache installs. It might be misleading, but all levels of ZTPServer logging will end up as an Apache error.

Example

```
[Fri Dec 12 10:49:42.186976 2014] [:error] [pid 864] INFO: [app:115] Logging started␣
↪for ztpserver
[Fri Dec 12 10:49:42.187112 2014] [:error] [pid 864] INFO: [app:116] Using repository␣
↪/usr/share/ztpserver
```

ZTPServer Configuration

Identify Nodes Based Upon Serial Number

Objective

I'd like the ZTPServer to use the switch's serial number for provisioning. This implies that all node directories in `nodes/` will be named using the serial number.

Solution

Open up the global ZTPServer configuration file:

```
admin@ztpserver:~# vi /etc/ztpserver/ztpserver.conf
```

Look for the line `identifier` and confirm it's set to `serialnumber`:

```
identifier = serialnumber
```

Restart the ztps process:

```
# If using Apache WSGI
admin@ztpserver:~# service apache2 restart

# If running in Standalone Mode, stop ztps
admin@ztpserver:~# pkill ztps

# Then start it again
admin@ztpserver:~# ztps
```

Explanation

The ZTPServer will use either the System MAC Address or the Serial Number of the switch as its System ID. The System ID is used to match statically provisioned nodes. Also, when a node is dynamically provisioned, the ZTPServer will create a new node directory for it in `nodes/` and it will be named using the System ID.

Identify Nodes Based Upon System MAC Address

Objective

I'd like the ZTPServer to use the switch's System MAC Address for provisioning. This implies that all node directories in `nodes/` will be named using the System MAC Address.

Solution

Open up the global ZTPServer configuration file:

```
admin@ztpserver:~# vi /etc/ztpserver/ztpserver.conf
```

Look for the line `identifier` and confirm it's set to `systemmac`:

```
identifier = systemmac
```

Restart the ztps process:

```
# If using Apache WSGI
admin@ztpserver:~# service apache2 restart

# If running in Standalone Mode, stop ztps
admin@ztpserver:~# pkill ztps

# Then start it again
admin@ztpserver:~# ztps
```

Explanation

The ZTPServer will use either the System MAC Address or the Serial Number of the switch as its System ID. The System ID is used to match statically provisioned nodes. Also, when a node is dynamically provisioned, the ZTPServer will create a new node directory for it in `nodes/` and it will be named using the System ID.

Enable/Disable Topology Validation

Objective

Topology Validation uses LLDP Neighbor information to make sure you have everything wired up correctly. Topology Validation is enabled/disabled in the main `ztpserver.conf` configuration file.

Solution

Open up the global ZTPServer configuration file:

```
admin@ztpserver:~# vi /etc/ztpserver/ztpserver.conf
```

Look for the line `disable_topology_validation`

```
# To disable Topology Validation
disable_topology_validation = True

# To enable Topology Validation
disable_topology_validation = False
```

Restart the ztps process:

```
# If using Apache WSGI
admin@ztpserver:~# service apache2 restart
```

```
# If running in Standalone Mode, stop ztps
admin@ztpserver:~# pkill ztps

# Then start it again
admin@ztpserver:~# ztps
```

Explanation

This configuration option enables/disables Topology Validation. This feature is extremely powerful and can help you confirm all of your nodes are wired up correctly. See the recipes under *Topology Validation* to learn more about the flexibility of Topology Validation.

Running the ZTPServer

Standalone - Change the ZTPServer Interface

Objective

I only want the ZTPServer process to listen on a specific network interface.

Solution

Open up the global ZTPServer configuration file:

```
admin@ztpserver:~# vi /etc/ztpserver/ztpserver.conf
```

Look for the line `interface` in the [server] group.

```
# To listen on all interfaces
interface = 0.0.0.0

# To listen on a specific interface
interface = 192.0.2.100
```

Restart the ztps process:

```
# If running in Standalone Mode, stop ztps
admin@ztpserver:~# pkill ztps

# Then start it again
admin@ztpserver:~# ztps &
```

Explanation

This recipe helps you define a specific interface for the ZTPServer to listen on.

Note: Be sure the `interface` coincides with the `server_url` value in the configuration file.

Standalone - Run ZTPServer on a Specific Port

Objective

I want to define which port the ZTPServer listens on.

Solution

Open up the global ZTPServer configuration file:

```
admin@ztpserver:~# vi /etc/ztpserver/ztpserver.conf
```

Look for the line `port` in the [server] group.

```
# Choose a port of your liking
port = 8080
```

Restart the ztps process:

```
# If running in Standalone Mode, stop ztps
admin@ztpserver:~# pkill ztps

# Then start it again
admin@ztpserver:~# ztps &
```

Explanation

This recipe helps you define a specific port for the ZTPServer to listen on.

Note: Be sure the `port` coincides with the `server_url` value in the configuration file.

Standalone - Run ZTPServer in a Sub-directory

Objective

I don't want to run the ZTPServer at the root of my domain, I want it in a sub-directory.

Solution

Open up the global ZTPServer configuration file:

```
admin@ztpserver:~# vi /etc/ztpserver/ztpserver.conf
```

Look for the line `server_url` in the [default] group.

```
# Choose a subdirectory
server_url = http://ztpserver:8080/not/in/root/anymore
```

Restart the ztps process:

```
# If running in Standalone Mode, stop ztps
admin@ztpserver:~# pkill ztps

# Then start it again
admin@ztpserver:~# ztps &
```

Explanation

The `server_url` key defines where the REST API lives. You do not need to change any of your file locations to affect change. Simply change the key above.

Note: You can confirm the change by doing a simple `wget http://server:port/new/directory/path/bootstrap` to retrieve the bootstrap script.

Apache - Run ZTPServer on a Specific Port

Objective

I'm running ZTPServer as a WSGI with Apache and want to change what port it listens on.

Solution

Apache configurations can vary widely, and the ZTPServer has no control over this, so view this simply as a suggestion.

Open up your Apache configuration file:

```
# Apache
admin@ztpserver:~# vi /etc/apache2/sites-enabled/ztpserver.conf

# HTTPd
admin@ztpserver:~# vi /etc/httpd/conf.d/ztpserver.conf
```

Change the `Listen` and `VirtualHost` values to the desired port.

```
LoadModule wsgi_module modules/mod_wsgi.so
Listen 8080

<VirtualHost *:8080>

    WSGIDaemonProcess ztpserver user=www-data group=www-data threads=50
    WSGIScriptAlias / /etc/ztpserver/ztpserver.wsgi
    # Required for RHEL
    #WSGISocketPrefix /var/run/wsgi

    <Location />
        WSGIProcessGroup ztpserver
        WSGIApplicationGroup %{GLOBAL}

        # For Apache <= 2.2, use Order and Allow
        Order deny,allow
        Allow from all
        # For Apache >= 2.4, Allow is replaced by Require
        Require all granted
    </Location>

    # Override default logging locations for Apache
    #ErrorLog /path/to/ztpserver_error.log
    #CustomLog /path/to/ztpserver_access.log
</VirtualHost>
```

Restart the ztps process:

```
# Restart Apache
admin@ztpserver:~# service apache2 restart
```

Explanation

When you run ZTPServer as a WSGI under Apache or like server, the interface and port that are used for listening for HTTP requests are controlled by the web server. The config snippet above shows how this might be done with Apache, but note that variations might arise in your own environment.

Apache - Run ZTPServer in a Sub-directory

Objective

I'm running ZTPServer as a WSGI with Apache and I want to change the path that the REST API resides.

Solution

WSGI-compliant webserver configurations can vary widely, so here's a sample of how this is done with Apache.

Open up the global ZTPServer configuration file:

```
admin@ztpserver:~# vi /etc/ztpserver/ztpserver.conf
```

Look for the line `server_url` in the [default] group.

```
# Choose a subdirectory
server_url = http://ztpserver:8080/not/in/root/anymore
```

You might think that you have to change your Apache conf to move this to a sub-directory, but you don't. Your config should look like the block below. Note the `<Location />`.

```
LoadModule wsgi_module modules/mod_wsgi.so
Listen 8080

<VirtualHost *:8080>

    WSGIDaemonProcess ztpserver user=www-data group=www-data threads=50
    WSGIScriptAlias / /etc/ztpserver/ztpserver.wsgi
    # Required for RHEL
    #WSGISocketPrefix /var/run/wsgi

    <Location />
        WSGIProcessGroup ztpserver
        WSGIApplicationGroup %{GLOBAL}

        # For Apache <= 2.2, use Order and Allow
        Order deny,allow
        Allow from all
        # For Apache >= 2.4, Allow is replaced by Require
        Require all granted
    </Location>

    # Override default logging locations for Apache
    #ErrorLog /path/to/ztpserver_error.log
    #CustomLog /path/to/ztpserver_access.log
</VirtualHost>
```

Restart the ztps process:

```
# Restart Apache
admin@ztpserver:~# service apache2 restart
```

Explanation

It might seem counter-intuitive but the Apache configuration should use the `Location` directive to point at root. The desired change to the path is done by the ZTPServer `server_url` configuration value in `/etc/ztpserver/ztpserver.conf`.

Change ZTPServer File Ownership

Objective

I'd like all of the ZTPServer provisioning files to be owned by a particular user/group.

Note: This is most often needed when running the ZTPServer WSGI App and the apache user is unable to read/write to `/usr/share/ztpserver`.

Solution

```
admin@ztpserver:~# chown -R myUser:myGroup /usr/share/ztpserver
admin@ztpserver:~# chmod -R ug+rw /usr/share/ztpserver
```

Explanation

The shell commands listed above set ownership and permissions for the default data_root location `/usr/share/ztpserver`. Be mindful that if you are running the ZTPServer WSGI App, the mod_wsgi daemon user must be able to read/write to these files.

Note: When running the ZTPServer WSGI App, you should also check the ownership and permission of `/etc/ztpserver/ztpserver.wsgi`.

Apache - Configure SELinux Permissions

Objective

My server has SELinux enabled and I'd like to set the ZTPServer file type so that Apache can read/write files in the data_root.

Note: This is most often needed when running the ZTPServer WSGI App and the apache user is unable to read/write to `/usr/share/ztpserver`.

Solution

```
# For Fedora - httpd
admin@ztpserver:~# chcon -Rv --type=httpd_sys_script_rw_t /usr/share/ztpserver

# For Ubuntu - Apache
admin@ztpserver:~# chcon -R -h system_u:object_r:httpd_sys_script_rw_t /usr/share/
↪ztpserver
```

Explanation

The shell commands listed above set the SELinux file attributes so that Apache can read/write to the files. This is often the case since `/usr/share/ztpserver` is not in the normal operating directory `/var/www/`. Note that the commands above are suggestions and you might consider tweaking them to suit your own environment.

Hello World - A Simple Provisioning Example

- *Prepare Your Switch for Provisioning*

Introduction

The following set of recipes will help you perform a basic provisioning task using the ZTPServer. There are some assumptions:

- You have already installed the ZTPServer
- You have performed the basic configuration to define which interface and port the server will run on.
- You have a DHCP server running with `option bootfile-name "http://<ZTPSERVER-URL>:<PORT>/bootstrap";` Sample config
- Your test (v)EOS node can receive DHCP responses
- Make sure the ztps process is not running

Note: If you would like to test this in a virtual environment, please see the packer-ztpserver Github repo to learn how to automatically install a ZTPServer with all of the complementary services (DHCP, DNS, NTP, XMPP, and SYSLOG). Both Virtual Box and VMware are supported.

Prepare Your Switch for Provisioning

Objective

I want to prepare my test device (vEOS or EOS) for use with the ZTPServer. This will put your switch into ZTP Mode, so backup any configs you want to save.

Solution

Log into your (v)EOS node, then:

```
switch-name> enable
switch-name# write erase
Proceed with erasing startup configuration? [confirm] y
switch-name# reload now
```

Explanation

ZTP Mode is enabled when a switch boots and there is no startup-config (or it's empty) found in `/mnt/flash/`. Therefore, we use the `write erase` command to clear the current startup-config and use `reload now` to reboot the switch. When the switch comes up you will see it enter ZTP Mode and begin sending DHCP requests on all interfaces.

Add a Static Node Entry

Objective

I want to provision my switch based upon its System MAC Address.

Solution

Log into your (v)EOS node to get its MAC Address. If it's in ZTP Mode, just log in with username `admin`:

```
switch-name> show version
```

Note: Copy the System MAC Address for later.

Confirm your ZTPServer Configuration will identify a node based upon its MAC:

```
admin@ztpserver:~# vi /etc/ztpserver/ztpserver.conf
```

Look for the line `identifier` and confirm it's set to `systemmac`:

```
identifier = systemmac
```

Finally, let's create a nodes directory for this device:

```
# Go to your data_root - by default it's /usr/share/ztpserver
admin@ztpserver:~# cd /usr/share/ztpserver

# Move to the nodes directory, where all node information is stored
admin@ztpserver:~# cd nodes

# Create a directory using the MAC Address you found earlier
admin@ztpserver:~# mkdir 001122334455
```

Explanation

A node is considered to be statically provisioned when a directory with its System ID is already located in the `nodes/` directory.

Note that the System ID can be the node's System MAC Address or its Serial Number. In this case we chose to use the `systemmac` since vEOS nodes don't have a Serial Number by default.

Just adding this directory is not enough to provision the node. The remaining recipes will finish off the task.

Create a Startup-Config with Minimal Configuration

Objective

When my node is provisioned, I want it to be passed a static startup-config. This config will include some basic Management network info including syslog and ntp. It will set the admin user's password to admin, and enable eAPI.

Solution

```
# Go to your data_root - by default it's /usr/share/ztpserver
admin@ztpserver:~# cd /usr/share/ztpserver

# Move to the specific node directory that you created earlier
admin@ztpserver:~# cd nodes/001122334455

# Create a startup-config
admin@ztpserver:~# vi startup-config
```

Copy and paste this startup-config, changing values where you see fit:

```
!
hostname test-node-1
ip name-server vrf default <DNS-SERVER-IP>
!
ntp server <NTP-SERVER-IP>
!
username admin privilege 15 role network-admin secret admin
!
interface Management1
 ip address <MGMT-IP-ADDRESS>/<SUBNET>
!
ip access-list open
 10 permit ip any any
!
ip route 0.0.0.0/0 <DEFAULT-GW>
!
ip routing
!
management api http-commands
 no shutdown
!
banner login
Welcome to $(hostname)!
This switch has been provisioned using the ZTPServer from Arista Networks
Docs: http://ztpserver.readthedocs.org/
Source Code: https://github.com/arista-eosplus/ztpserver
EOF
!
end
```

Explanation

When the ZTPServer receives a request from your node to begin provisioning, it will find the directory `nodes/001122334455` and know that this node is statically configured. In this case, a `startup-config` must be present. In practice, the ZTPServer tells the node to perform the `config_replace` action with this file as the source.

Add Event Handler to Backup the startup-config to the ZTPServer

Objective

I want to backup the latest startup-config from my node so that if I make changes or have to replace the node I have the latest copy.

Note: By adding this, the node will perform an HTTP PUT and overwrite the `nodes/001122334455/startup-config` file.

Solution

```
# Go to your data_root - by default it's /usr/share/ztpserver
admin@ztpserver:~# cd /usr/share/ztpserver

# Move to the specific node directory that you created earlier
admin@ztpserver:~# cd nodes/001122334455

# Edit your startup-config
admin@ztpserver:~# vi startup-config
```

Add the following lines to your startup-config, changing values where needed:

```
event-handler configpush
 trigger on-startup-config
 ! For default VRF, make sure to update the ztpserver url
 action bash export SYSMAC=`FastCli -p 15 -c 'show ver | grep MAC | cut -d" " -f 5' |␣
↪sed 's/[.]*//g'`; curl http://<ZTPSERVER-URL>:<PORT>/nodes/$SYSMAC/startup-config -
↪H "content-type: text/plain" --data-binary @/mnt/flash/startup-config -X PUT
 ! For non-default VRF, update and use:
 ! action bash export SYSMAC=`FastCli -p 15 -c 'show ver | grep MAC | cut -d" " -f 5'␣
↪| sed 's/[.]*//g'`; ip netns exec ns-<VRF-NAME> curl http://<ZTPSERVER-URL>:<PORT>/
↪nodes/$SYSMAC/startup-config -H "content-type: text/plain" --data-binary @/mnt/
↪flash/startup-config -X PUT
```

Explanation

By adding this line to the startup-config, this configuration will be sent down to the node during provisioning. From that point onward, the node will perform and HTTP PUT of the startup-config and the ZTPServer will overwrite the startup-config file in the node's directory.

Install a Specific (v)EOS Version

Objective

I want a specific (v)EOS version to be automatically installed when I provision my node.

Note: This assumes that you've already downloaded the desired (v)EOS image from Arista.

Solution

Let's create a place on the ZTPServer to host some SWIs:

```
# Go to your data_root - by default it's /usr/share/ztpserver
admin@ztpserver:~# cd /usr/share/ztpserver

# Create an images directory
admin@ztpserver:~# mkdir -p files/images

# SCP your SWI into the images directory, name it whatever you like
admin@ztpserver:~# scp admin@otherhost:/tmp/vEOS.swi files/images/vEOS_4.14.5F.swi
```

Now let's create a definition that performs the `install_image` action:

```
# Go to your data_root - by default it's /usr/share/ztpserver
admin@ztpserver:~# cd /usr/share/ztpserver

# Move to the specific node directory that you created earlier
admin@ztpserver:~# cd nodes/001122334455

# Create a definition file
admin@ztpserver:~# vi definition
```

Add the following lines to your definition, changing values where needed:

```
---
name: static node definition
actions:
  -
    action: install_image
    always_execute: true
    attributes:
      url: files/images/vEOS_4.14.5F.swi
      version: 4.14.5F
    name: "Install 4.14.5F"
```

Note: The definition uses YAML syntax

Explanation

The definition is where we list all of the actions we want the node to execute during the provisioning process. In this case we are hosting the SWI on the ZTPServer, so we just define the `url` in relation to the `data_root`. We could change the `url` to point to another server altogether - the choice is yours. The benefit in hosting the file on the ZTPServer is that we perform an extra checksum step to validate the integrity of the file.

In practice, the node requests its definition during the provisioning process. It sees that it's supposed to perform the `install_image` action, so it requests the `install_image` python script. It then performs an HTTP GET for the `url`. Once it has these locally, it executes the `install_image` script.

Start ZTPServer in Standalone Mode

Objective

Okay, enough reading and typing; let's push some buttons!

Solution

Let's run the ZTPServer in Standalone Mode since this is just a small test. Login to your ZTPServer:

```
# Start the ZTPServer - console loggin will appear
admin@ztpserver:~# ztps
INFO: [app:115] Logging started for ztpserver
INFO: [app:116] Using repository /usr/share/ztpserver
Starting server on http://<ZTPSERVER-URL>:<PORT>
```

Explanation

The easiest way to run the ZTPServer is in Standalone Mode - which is done by typing `ztps` in a shell. This will cause the configured interface and port to start listening for HTTP requests. Your DHCP server will provide the node with `option bootfile-name "http://<ZTPSERVER-URL>:<PORT>/bootstrap"` in the DHCP response, which lets the node know where to grab the bootstrap script.

A Quick Overview of the Provisioning Process for this Node

1. **GET /bootstrap**: The node gets the bootstrap script and begins executing it. The following requests are made while the bootstrap script is being executed.
2. **GET /bootstrap/config**: The node gets the bootstrap config which contains XMPP and Syslog information for the node to send logs to.
3. **POST /nodes**: The node sends information about itself in JSON format to the ZTPServer. The ZTPServer parses this info and finds the System MAC. It looks in the `nodes/` directory and finds a match.
4. **GET /nodes/001122334455**: The node requests its definition and learns what resources it has to retrieve.
5. **GET /actions/install_image**: The node retrieves the install_image script.
6. **GET /files/images/vEOS_4.14.5F.swi**: The node retrieves the SWI referenced in the definition.
7. **GET /meta/files/images/vEOS_4.14.5F.swi**: The node retrieves the checksum of the SWI for validation and integrity.
8. **GET /actions/replace_config**: The node retrieves the replace_config script.
9. **GET /nodes/001122334455/startup-config**: The node retrieves the startup-config we created earlier.
10. **GET /meta/nodes/001122334455/startup-config**: The node retrieves the checksum of the startup-config.
11. **Node Applies Config and Reboots**
12. **PUT /nodes/001122334455/startup-config**: The node uploads its current startup-config.

Provision a Static Node

- *Add a Static Node Entry*
- *Create a Startup-Config File*

Add a Static Node Entry

Objective

I want to provision my switch based upon its System ID (System MAC Address or Serial Number).

Solution

Log into your (v)EOS node to get its System ID. If it's in ZTP Mode, just log in with username `admin`:

```
switch-name> show version
```

Note: Copy down the System ID (System MAC Address or Serial Number).

Let's create a node directory for this device:

```
# Go to your data_root - by default it's /usr/share/ztpserver
admin@ztpserver:~# cd /usr/share/ztpserver

# Move to the nodes directory, where all node information is stored
admin@ztpserver:~# cd nodes

# Create a directory using the MAC Address you found earlier
admin@ztpserver:~# mkdir <SYSTEM_ID>
```

Explanation

A node is considered to be statically provisioned when a directory with its System ID is already located in the `nodes/` directory.

Note that the System ID can be the node's System MAC Address or its Serial Number.

Just adding this directory is not enough to provision the node. The remaining recipes will finish off the task. To successfully provision a node statically, you will need to create:

- `startup-config`
- `pattern` file - if Topology Validation is enabled
- `definition` - if you choose to apply other actions during provisioning

and place them in `[data_root]/nodes/<SYSTEM_ID>`.

Note: Confirm your ZTPServer Configuration will identify a node based upon the desired System ID by checking /etc/ztpserver/ztpserver.conf and check the value of `identifier`

Create a Startup-Config File

Objective

I want the node to receive a startup-config during provisioning.

Solution

Create a file named `startup-config` in `[data_root]/nodes/<SYSTEM_ID>/`.

Go to your data_root - by default it's /usr/share/ztpserver admin@ztpserver:~# cd /usr/share/ztpserver

Move to the node directory you created above. admin@ztpserver:~# cd nodes/<SYSTEM_ID>

Create/edit the startup-config file admin@ztpserver:~# vi startup-config

Place the desired configuration into the startup-config. Here's an example. Please change values where you see fit:

```
!
hostname test-node-1
ip name-server vrf default <DNS-SERVER-IP>
!
ntp server <NTP-SERVER-IP>
!
username admin privilege 15 role network-admin secret admin
!
interface Management1
 ip address <MGMT-IP-ADDRESS>/<SUBNET>
!
ip access-list open
 10 permit ip any any
!
ip route 0.0.0.0/0 <DEFAULT-GW>
!
ip routing
!
management api http-commands
 no shutdown
!
banner login
Welcome to $(hostname)!
This switch has been provisioned using the ZTPServer from Arista Networks
Docs: http://ztpserver.readthedocs.org/
Source Code: https://github.com/arista-eosplus/ztpserver
EOF
!
end
```

Explanation

A startup-config file is required when you statically provision a node. The format of the startup-config is the same as you are used to, which can be found on your switch at file:startup-config (/mnt/flash/startup-config)

Create a Pattern (Topology Validation enabled)

Objective

I have created a static node directory and Topology Validation is enabled, so I would like to make sure everything is wired up correctly before provisioning a node.

Note: YAML syntax can be a pain sometimes. The indentation is done with spaces and not tabs.

Solution

Create a file named `pattern` in `[data_root]/nodes/<SYSTEM_ID>/` and define the LLDP associations.

> # Go to your data_root - by default it's /usr/share/ztpserver admin@ztpserver:~# cd /usr/share/ztpserver
>
> # Move to the node directory you created above. admin@ztpserver:~# cd nodes/<SYSTEM_ID>
>
> # Create/edit the pattern file admin@ztpserver:~# vi pattern

Example 1: Match any neighbor

This pattern essentially disables Topology Validation.

```
---
name: Match anything
interfaces:
  - any: any:any
```

Example 2: Match any interface on a specific neighbor

This pattern says, the node being provisioned must be connected to a neighbor with hostname `pod1-spine1` but it can be connected to any peer interface.

```
---
name: Anything on pod1-spine1
interfaces:
  - any: pod1-spine1:any
```

Example 3: Match specific interface on a specific neighbor

This pattern says, the node being provisioned must be connected to a neighbor with hostname `pod1-spine1` on Ethernet1.

```
---
name: Anything on pod1-spine1
interfaces:
  - any: pod1-spine1:Ethernet1
```

Example 4: Make sure I'm not connected to a node

This pattern is the same as Example #2, but we add another check to make sure the node being provisioned is not connected to any spines in `pod2`.

```
---
name: Not connected to anything in pod2
interfaces:
  - any: pod1-spine1:any
  - any: regex('pod2-spine\d+'):none
  - none: regex('pod2-spine\d+'):any #equivalent to line above
```

Example 5: Using variables in the pattern

This pattern is similar to what you've seen above except we use variables to make things easier.

```
---
name: Not connected to any spine in pod2
variables:
  - not_pod2: regex('pod2-spine\d+')
interfaces:
  - any: pod1-spine1:any
  - any: $not_pod2:none
```

Explanation

Pattern files are YAML-based and are the underpinnings of Topology Validation. A node will not be successfully provisioned if it cannot pass all of the interface tests contained within the pattern file. The examples above are just a small sample of the complex associations you can create. Take a look at the neighbordb section to learn more.

Note: YAML can be a pain, and invalid YAML syntax will cause provisioning to fail. You can make sure your syntax is correct by using a tool like YAMLlint

---

Create a Definition File

Objective

Aside from sending the node a startup-config, I'd like to upgrade the node to a specific v(EOS) version.

Solution

These types of system changes are accomplished via the `definition` file. The definition is a YAML-based file with a section for each action that you want to execute.

Note: Learn more about Actions.

```
# Go to your data_root - by default it's /usr/share/ztpserver
admin@ztpserver:~# cd /usr/share/ztpserver

# Create an images directory
```

```
admin@ztpserver:~# mkdir -p files/images

# SCP your SWI into the images directory, name it whatever you like
admin@ztpserver:~# scp admin@otherhost:/tmp/vEOS.swi files/images/vEOS_4.14.5F.swi
```

Now let's create a definition that performs the `install_image` action:

```
# Go to your data_root - by default it's /usr/share/ztpserver
admin@ztpserver:~# cd /usr/share/ztpserver

# Move to the specific node directory that you created earlier
admin@ztpserver:~# cd nodes/<SYSTEM_ID>

# Create a definition file
admin@ztpserver:~# vi definition
```

Add the following lines to your definition, changing values where needed:

```
---
name: static node definition
actions:
  -
    action: install_image
    always_execute: true
    attributes:
      url: files/images/vEOS_4.14.5F.swi
      version: 4.14.5F
    name: "Install 4.14.5F"
```

Explanation

The definition is where we list all of the actions we want the node to execute during the provisioning process. In this case we are hosting the SWI on the ZTPServer, so we just define the `url` in relation to the `data_root`. We could change the `url` to point to another server altogether - the choice is yours. The benefit in hosting the file on the ZTPServer is that we perform an extra checksum step to validate the integrity of the file.

In practice, the node requests its definition during the provisioning process. It sees that it's supposed to perform the `install_image` action, so it requests the `install_image` python script. It then performs an HTTP GET for the `url`. Once it has these locally, it executes the `install_image` script.

Create an Attributes File

Objective

I want to use variables in my definition and abstract the values to a unique file. These variables will be sent down to the node during provisioning and be used while the node is executing the actions listed in the definition.

Solution

Create a file named `attributes` in `[data_root]/nodes/<SYSTEM_ID>/`.

Go to your data_root - by default it's /usr/share/ztpserver admin@ztpserver:~# cd /usr/share/ztpserver

Move to the node directory you created above. admin@ztpserver:~# cd nodes/<SYSTEM_ID>

Move to the node directory you created above. admin@ztpserver:~# vi attributes

Here's the different type of ways to define the attributes:

Example 1: A simple key/value pair

```
---
ntp_server: ntp.example.com
dns_server: ns1.example.com
```

Example 2: key/dictionary

```
---
system_config:
  ntp: ntp.example.com
  dns: ns1.example.com
```

Example 3: key/list (note the hyphens)

```
---
dns_servers:
  - ns1.example.com
  - ns2.example.com
  - ns3.example.com
  - ns4.example.com
```

Example 4: Referencing another variable

```
---
ntp_server: ntp.example.com
other_var: $ntp_server
```

Borrowing from the definition recipe above, we can replace some values with variables from the attributes file:

nodes/<SYSTEM_ID>/definition

```
---
name: static node definition
actions:
  -
    action: install_image
    always_execute: true
    attributes:
      url: $swi_url
      version: $swi_version
    name: $swi_name
```

and the **nodes/<SYSTEM_ID>/attributes**

```
---
swi_url: files/images/vEOS_4.14.5F.swi
swi_version: 4.14.5F
swi_name: "Install 4.14.5F"
```

Explanation

The `attributes` file is optional. The variables that are contained within it are sent to the node during provisioning. In the final example above you can see how the attributes file and definition work in concert. Note that the ZTPServer

performs variable substitution when the node requests the definition via GET /nodes/<SYSTEM_ID>. By removing the static values from the definition, we can use the same definition for multiple nodes (using symlink) and just create unique attributes files in the node's directory.

It's important to note that these variables can exist in different places and accomplish the same task. In this recipe we created a unique attributes file, which lives in the node's directory. You can also put these attributes directly into the definition file like the example below.

Example: At the global scope of the definition

```
---
name: static node definition
actions:
  -
    action: install_image
    always_execute: true
    attributes:
      url: $swi_url
      version: $swi_version
    name: $swi_name
attributes:
  swi_url: files/images/vEOS_4.14.5F.swi
  swi_version: 4.14.5F
  swi_name: "Install 4.14.5F"
```

Symlink to a Generic Definition

Objective

I'd like to use the same definition for multiple static node directories without manually updating each one.

Solution

Create one definition in the `[data_root]/definitions` folder and create a symlink to the specific `[data_root]/nodes/<SYSTEM_ID>/` folder.

"[data_root]/definitions/static_node_definition

```
---
name: static node definition
actions:
  -
    action: install_image
    always_execute: true
    attributes:
      url: $swi_url
      version: $swi_version
    name: $swi_name
```

and the **nodes/<SYSTEM_ID>/attributes**

```
---
swi_url: files/images/vEOS_4.14.5F.swi
swi_version: 4.14.5F
swi_name: "Install 4.14.5F"
```

then create the symlink

```
# Go to your node's unique directory
admin@ztpserver:~# cd /usr/share/ztpserver/nodes/<SYSTEM_ID>

# Create the symlink
admin@ztpserver:~# ln -s /usr/share/ztpserver/definitions/static_node_definition ./
↪definition
```

Explanation

The steps above let you reuse a single definition file for many static nodes. Note that the variables are located in the attributes file in the `nodes/<SYSTEM_ID>/` folder.

Provision a Dynamic Node

Using Open Patterns

Objective

I want to provision a node without knowing anything about it. I just want it to receive a default configuration.

Solution

You can accomplish this by using neighbordb. Neighbordb contains associations between LLDP neighbor patterns and definitions. So if we use a pattern that matches anything, we can use it to assign a simple, default definition.

```
# Go to your data_root - by default it's /usr/share/ztpserver
admin@ztpserver:~# cd /usr/share/ztpserver

# Modify your neighbordb
admin@ztpserver:~# vi neighbordb
```

Add the following lines to your definition, changing values where needed:

```
---
patterns:
  - name: Default Pattern
    definition: default
    interfaces:
      - any: any:any
```

If you happen to be provisioning a node in isolation and the node does not have any neighbors, use the following pattern:

```
---
patterns:
  - name: Default Pattern
    definition: default
    interfaces:
      - none: none:none
```

Then add a definition to `[data_root]/definitions/default`

Note: See the sections on Definitions and Actions to learn more.

Explanation

By placing this pattern in your neighbordb, the ZTPServer will allow this node to be provisioned and will assign it the `default` definition. Use caution when placing this pattern in your neighbordb as it might allow nodes to receive the `default` definition when you intend them to receive another pattern.

Identify a Node Based Upon Specific Neighbor

Objective

I want my node to be dynamically provisioned based upon a specific LLDP neighbor association.

Solution

Modify your neighbordb:

```
# Go to your data_root - by default it's /usr/share/ztpserver
admin@ztpserver:~# cd /usr/share/ztpserver

# Modify your neighbordb
admin@ztpserver:~# vi neighbordb
```

Then add the pattern that includes the required match.

```
---
patterns:
  - name: tora for pod1
    definition: tora
    interfaces:
      - Ethernet1: dc1-pod1-spine1:Ethernet1
```

This pattern says that the node being provisioned must have a connection between its Ethernet1 and dc1-pod1-spine1's Ethernet1.

Explanation

In this recipe we use neighbordb to link a pattern with a definition. When a node executes the bootstrap script it will send the ZTPServer some information about itself. The ZTPServer will not find any existing directory with the node's System-ID (System MAC or Serial Number depending upon your configuration) so it next checks neighbordb to try

and find a match. The ZTPServer will analyze the nodes LLDP neighbors, find the match in neighbordb and then apply the `tora` definition.

Identify a Node's Neighbors Using Regex

Objective

I want my node to be dynamically provisioned and I'd like to match certain neighbors using regex.

Solution

Modify your neighbordb:

```
# Go to your data_root - by default it's /usr/share/ztpserver
admin@ztpserver:~# cd /usr/share/ztpserver

# Modify your neighbordb
admin@ztpserver:~# vi neighbordb
```

Then add the pattern that includes the required match.

```
---
patterns:
  - name: tora for pod1
    definition: tora
    interfaces:
      - Ethernet1: regex('dc1-pod1-spine\D+'):Ethernet1
```

This pattern says that the node being provisioned must have a connection between its Ethernet1 and any dc1-pod1-spines Ethernet1.

Explanation

In this recipe we use neighbordb to link a pattern with a definition. When a node executes the bootstrap script it will send the ZTPServer some information about itself. The ZTPServer will not find any existing directory with the node's System-ID (System MAC or Serial Number depending upon your configuration) so it next checks neighbordb to try and find a match. The ZTPServer will analyze the nodes LLDP neighbors, find the match in neighbordb and then apply the `tora` definition.

Note: There are a few different functions that you can use other than `regex()`. Check out this section to learn more.

Topology Validation

Enable/Disable Topology Validation

Objective

Topology Validation uses LLDP Neighbor information to make sure you have everything wired up correctly. Topology Validation is enabled/disabled in the main `ztpserver.conf` configuration file.

Solution

Open up the global ZTPServer configuration file:

```
admin@ztpserver:~# vi /etc/ztpserver/ztpserver.conf
```

Look for the line `disable_topology_validation`

```
# To disable Topology Validation
disable_topology_validation = True

#To enable Topology Validation
disable_topology_validation = False
```

Restart the ztps process:

```
# If using Apache WSGI
admin@ztpserver:~# service apache2 restart

# If running in Standalone Mode, stop ztps
admin@ztpserver:~# pkill ztps

# Then start it again
admin@ztpserver:~# ztps
```

Explanation

This configuration option enables/disables Topology Validation. This feature is extremely powerful and can help you confirm all of your nodes are wired up correctly. See the recipes below to learn more about the flexibility of Topology Validation.

Allow Any Neighbor

Objective

I want to provision a node without knowing anything about it. I just want it to receive a default configuration.

Solution

You can accomplish this by using neighbordb. Neighbordb contains associations between LLDP neighbor patterns and definitions. So if we use a pattern that matches anything, we can use it to assign a simple, default definition.

```
# Go to your data_root - by default it's /usr/share/ztpserver
admin@ztpserver:~# cd /usr/share/ztpserver

# Modify your neighbordb
admin@ztpserver:~# vi neighbordb
```

Add the following lines to your definition, changing values where needed:

```
---
patterns:
  - name: Default Pattern
    definition: default
    interfaces:
      - any: any:any
```

If you happen to be provisioning a node in isolation and the node does not have any neighbors, use the following pattern:

```
---
patterns:
  - name: Default Pattern
    definition: default
    interfaces:
      - none: none:none
```

Then add a definition to `[data_root]/definitions/default`

Note: See the sections on Definitions and Actions to learn more.

Explanation

By placing this pattern in your neighbordb, the ZTPServer will allow this node to be provisioned and will assign it the `default` definition. Use caution when placing this pattern in your neighbordb as it might allow nodes to receive the `default` definition when you intend them to receive another pattern.

Match Pattern with Exact String

Objective

I want my node to be dynamically provisioned based upon a specific LLDP neighbor association.

Solution

Modify your neighbordb:

```
# Go to your data_root - by default it's /usr/share/ztpserver
admin@ztpserver:~# cd /usr/share/ztpserver

# Modify your neighbordb
admin@ztpserver:~# vi neighbordb
```

Then add the pattern that includes the required match.

```
---
patterns:
  - name: tora for pod1
    definition: tora
    interfaces:
      - Ethernet1: dc1-pod1-spine1:Ethernet1
```

This pattern says that the node being provisioned must have a connection between its Ethernet1 and dc1-pod1-spine1's Ethernet1.

Explanation

In this recipe we use neighbordb to link a pattern with a definition. When a node executes the bootstrap script it will send the ZTPServer some information about itself. The ZTPServer will not find any existing directory with the node's System-ID (System MAC or Serial Number depending upon your configuration) so it next checks neighbordb to try and find a match. The ZTPServer will analyze the nodes LLDP neighbors, find the match in neighbordb and then apply the `tora` definition.

Match Pattern Using a Regular Expression

Objective

I want my node to be dynamically provisioned and I'd like to match certain neighbors using regex.

Solution

Modify your neighbordb:

```
# Go to your data_root - by default it's /usr/share/ztpserver
admin@ztpserver:~# cd /usr/share/ztpserver

# Modify your neighbordb
admin@ztpserver:~# vi neighbordb
```

Then add the pattern that includes the required match.

```
---
patterns:
  - name: tora for pod1
    definition: tora
    interfaces:
      - Ethernet1: regex('dc1-pod1-spine\D+'):Ethernet1
```

This pattern says that the node being provisioned must have a connection between its Ethernet1 and any dc1-pod1-spines Ethernet1.

Explanation

In this recipe we use neighbordb to link a pattern with a definition. When a node executes the bootstrap script it will send the ZTPServer some information about itself. The ZTPServer will not find any existing directory with the node's System-ID (System MAC or Serial Number depending upon your configuration) so it next checks neighbordb to try and find a match. The ZTPServer will analyze the nodes LLDP neighbors, find the match in neighbordb and then apply the `tora` definition.

Note: There are a few different functions that you can use other than `regex()`. Check out this section to learn more.

Match Pattern That Includes a String

Objective

I want my node to be dynamically provisioned and I'd like to match certain neighbors as long as the neighbor hostname includes a certain string.

Solution

Modify your neighbordb:

```
# Go to your data_root - by default it's /usr/share/ztpserver
admin@ztpserver:~# cd /usr/share/ztpserver

# Modify your neighbordb
admin@ztpserver:~# vi neighbordb
```

Then add the pattern that includes the required match.

```
---
patterns:
  - name: tora for pod1
    definition: tora
    interfaces:
      - Ethernet1: includes('dc1-pod1'):Ethernet1
```

This pattern says that the node being provisioned must have a connection between its Ethernet1 and any hostname that includes `dc1-pod1` Ethernet1.

Explanation

In this recipe we use neighbordb to link a pattern with a definition. When a node executes the bootstrap script it will send the ZTPServer some information about itself. The ZTPServer will not find any existing directory with the node's System-ID (System MAC or Serial Number depending upon your configuration) so it next checks neighbordb to try and find a match. The ZTPServer will analyze the nodes LLDP neighbors, find the match in neighbordb and then apply the `tora` definition.

Match Pattern That Excludes a String

Objective

I want my node to be dynamically provisioned and I'd like to match certain neighbors as long as the neighbor hostname excludes a certain string.

Solution

Using the `excludes()` function allows you to match the inverse of the `includes()` function.

Modify your neighbordb:

```
# Go to your data_root - by default it's /usr/share/ztpserver
admin@ztpserver:~# cd /usr/share/ztpserver

# Modify your neighbordb
admin@ztpserver:~# vi neighbordb
```

Then add the pattern that includes the required match.

```
---
patterns:
  - name: tora for pod1
    definition: tora
    interfaces:
      - Ethernet1: includes('dc1-pod1'):Ethernet1
      - any: excludes('spine'):Ethernet50
```

This pattern says that the node being provisioned must have a connection between its Ethernet1 and any hostname that includes `dc1-pod1` Ethernet1.

Explanation

In this recipe we use neighbordb to link a pattern with a definition. When a node executes the bootstrap script it will send the ZTPServer some information about itself. The ZTPServer will not find any existing directory with the node's System-ID (System MAC or Serial Number depending upon your configuration) so it next checks neighbordb to try and find a match. The ZTPServer will analyze the nodes LLDP neighbors, find the match in neighbordb and then apply the `tora` definition.

Definitions

Add an Action to a Definition

Objective

I want to use one of the built-in actions in my definition file.

Solution

You can choose any of the pre-built actions to include in your definition.

Note: Learn more about Actions.

In this example we'll copy a python script to the node and set its permissions.

```
---
actions:
  -
    action: copy_file
    always_execute: true
    attributes:
      dst_url: /mnt/flash/
      mode: 777
      overwrite: if-missing
      src_url: files/automate/bgpautoinf.py
    name: "automate BGP peer interface config"
```

Explanation

Here we add the `copy_file` action to our definition. The attributes listed in the action will be passed to the node so that it is able to retrieve the script from `[SERVER_URL]/files/automate/bgpautoinf.py`. Since we are using `overwrite: if-missing`, the action will only copy the file to the node if it does not already exist.

Note: For more Action recipes see the Actions section.

Add Global Variables to Definition

Objective

I want to use a variable throughout my definition without having to define it more than once.

Solution

You can accomplish this by adding an `attributes` section at the root level of your definition file.

Note: Learn more about Actions.

In this example, we have two different actions that reference the same `$mode` and `$dst` variables.

```
---
actions:
  -
    action: copy_file
    always_execute: true
    attributes:
      dst_url: $dst
      mode: $mode
      overwrite: if-missing
      src_url: files/automate/bgpautoinf.py
    name: "Copy automate BGP script to node"
  -
    action: copy_file
    always_execute: true
    attributes:
      dst_url: $dst
      mode: $mode
      overwrite: if-missing
      src_url: files/automate/superautomate.py
    name: "Copy awesome script to my node"
  -
    action: add_config
    attributes:
      url: files/templates/ma1.template
      variables:
        ipaddress: $ip
    name: "configure ma1"
  -
    action: add_config
    attributes:
      url: files/templates/xmpp.template
      variables: $variables
    name: "configure ma1"

attributes:
  dst: /mnt/flash
  mode: 777
  ip: 192.168.0.50
  variables:
    domain: im.example.com
    user: myXmmpUser
    passwd: secret
    room: myAwesomeRoom
```

Explanation

This example shows how to use global variables within the definition. It's important to see the difference between using variables to define attributes of the action versus variables that get used within the template in an `add_config` action. See how the `ipaddress` variable is nested within a `variables` key? Also, you can create a list in the `attributes` section and pass the entire list into the action as shown in the XMPP config action.

Note: For more Action recipes see the Actions section.

Add Custom Log Statements as Action Executes

Objective

I want to send specific messages to my syslog and/or XMPP servers while an action is executing. Especially, if something goes wrong, I'd like to add a helpful message so the engineer knows who to contact.

Solution

The node being provisioned will send predefined logs to the endpoints defined in `[data_root]/bootstrap/bootstrap.conf`, but you can send additional client-side logs by adding a few attributes to your definition.

Let's add some specific status messages to the definition below.

Note: This could be a static node definition in `[data_root]/nodes/<SYSTEM_ID>/definition` or a global definition in `[data_root]/definitions/definition_name`.

```
---
actions:
  -
    action: copy_file
    always_execute: true
    attributes:
      dst_url: $dst
      mode: $mode
      overwrite: if-missing
      src_url: files/automate/bgpautoinf.py
    name: "Copy automate BGP script to node"
    onstart: "Starting the action to copy the BGP script"
    onsuccess: "SUCCESS: The BGP script has been copied"
    onfailure: "ERROR: Failed to copy script - contact admin@example.com"
attributes:
  dst: /mnt/flash
  mode: 777
```

Explanation

Here we make use of three specific keywords: `onstart`, `onsuccess` and `onfailure`. By adding these keys to your definition, the node will generate this message while it is being provisioned. As mentioned above, this message will be sent to all of the logging destinations defined in `[data_root]/bootstrap/bootstrap.conf`.

Note: For help defining an XMPP or syslog endpoint, see *Client-Side Logging*

Actions

- *Add a Configuration Block to a Node*
- *Add Configuration to a Node Using Variables*

Add a Configuration Block to a Node

Objective

In order to keep your provisioning data modular, you may want to break apart configuration blocks into small code blocks. You can use the `add_config` action to place a block on code on the node.

Solution

Example 1: Add a static block of configuration to your node

First, create a template file with the desired configuration.

```
# Go to your data_root - by default it's /usr/share/ztpserver
admin@ztpserver:~# cd /usr/share/ztpserver

# Make sure you have a directory for templates
admin@ztpserver:~# mkdir -p files/templates

# Create a static config block
admin@ztpserver:~# vi files/templates/east-dns.template
```

```
!
ip name-server vrf default east.ns1.example.com
!
```

Then add the `add_config` action to your definition:

```
---
actions:
  -
    action: add_config
    attributes:
      url: files/templates/east-dns.template
    name: "Add East DNS Server"
```

Explanation

Here we defined a simple action that adds configuration to the node during provisioning. The `url` in this case is relative to `[data_root]`/url. It's important to realize that the ZTPServer does not compile these configuration blocks into a startup-config and then send a single file to the node. Rather, the node executes each action independently, building the configuration in a module fashion. If you are interested in performing variable substitution in your templates to make them more flexible, see the next recipe.

Note: Please see the add_config documentation for more details.

Add Configuration to a Node Using Variables

Objective

I want to keep my templates flexible by using variables. In some cases, I'd like to assign a variable from a resource pool.

Solution

First, create a template file with the desired configuration. In this recipe let's configure interface Management1.

```
# Go to your data_root - by default it's /usr/share/ztpserver
admin@ztpserver:~# cd /usr/share/ztpserver

# Make sure you have a directory for templates
admin@ztpserver:~# mkdir -p files/templates

# Create a static config block
admin@ztpserver:~# vi files/templates/ma1.template
```

Paste this config into the template:

```
!
interface Management1
  ip address $ipaddress
  no shutdown
!
```

Then add the `add_config` action to your definition:

```
---
actions:
  -
    action: add_config
    attributes:
      url: files/templates/ma1.template
      variables:
        ipaddress: allocate("mgmt_subnet")
    name: "Configure Ma1"
```

Then create a resource pool called mgmt_subnet:

```
# Create a resource pool
admin@ztpserver:~# vi resources/mgmt_subnet
```

Paste the following into `mgmt_subnet`:

```
192.0.2.10/24: null
192.0.2.11/24: null
192.0.2.12/24: null
192.0.2.13/24: null
```

Explanation

This recipe ties a few different concepts together. From a high-level, the definition contains an action, `add_config`, which references a configuration block, `ma1.template`. Further, we use a variable, `$ipaddress` in the template file so that the template can be used for all nodes being provisioned. The final piece is the use of the `allocate()` plugin, which dynamically assigns a key from the associated file-based resource pool.

In practice, when a node requests its definition the ZTPServer will execute the `allocate("mgmt_subnet")` plugin and assign a key from the pool. The ZTPServer will then write the SYSTEM_ID as the value, overwriting `null`.

If you wanted to use the assigned value elsewhere in the definition, simply call `allocate(mgmt_subnet)` and the plugin will not assign a new value, rather it will return the key already assigned. Note that this is an implementation-detail specific to this particular plugin - other plugins might vary (please read the associated documentation for each).

The result would look like:

```
192.0.2.10/24: <SYSTEM_ID>
192.0.2.11/24: null
192.0.2.12/24: null
192.0.2.13/24: null
```

Note: Please see the add_config documentation for more details.

Replace Entire Startup-Config During Provisioning

Objective

I have a complete startup-config that I want to apply during provisioning. I want to completely replace what's already on the node.

Solution

First, create the startup-config with the desired configuration.

```
# Go to your data_root - by default it's /usr/share/ztpserver
admin@ztpserver:~# cd /usr/share/ztpserver

# Make sure you have a directory for templates
admin@ztpserver:~# mkdir -p files/configs

# Create a startup-config
admin@ztpserver:~# vi files/configs/tor-startup-config
```

```
!
hostname test-node-1
ip name-server vrf default <DNS-SERVER-IP>
!
ntp server <NTP-SERVER-IP>
!
username admin privilege 15 role network-admin secret admin
!
interface Management1
```

```
 ip address <MGMT-IP-ADDRESS>/<SUBNET>
!
ip access-list open
 10 permit ip any any
!
ip route 0.0.0.0/0 <DEFAULT-GW>
!
ip routing
!
management api http-commands
 no shutdown
!
banner login
Welcome to $(hostname)!
This switch has been provisioned using the ZTPServer from Arista Networks
Docs: http://ztpserver.readthedocs.org/
Source Code: https://github.com/arista-eosplus/ztpserver
EOF
!
end
```

Then add the `replace_config` action to your definition:

```
---
actions:
  -
    action: replace_config
    attributes:
      url: files/configs/tor-startup-config
    name: "Replace entire startup-config"
```

Explanation

This action simply replaces the `startup-config` which lives in `/mnt/flash/startup-config`.

Note: Please see the replace_config documentation for more details.

Copy a File to a Node During Provisioning

Objective

I want to copy a file to the node during the provisioning process and then set its permissions.

Solution

In this example we'll copy a python script to the node and set its permissions.

```
---
actions:
  -
    action: copy_file
```

```
    always_execute: true
    attributes:
      dst_url: /mnt/flash/
      mode: 777
      overwrite: if-missing
      src_url: files/automate/bgpautoinf.py
    name: "automate BGP peer interface config"
```

Explanation

Here we add the `copy_file` action to our definition. The attributes listed in the action will be passed to the node so that it is able to retrieve the script from `[SERVER_URL]/files/automate/bgpautoinf.py`. Since we are using `overwrite: if-missing`, the action will only copy the file to the node if it does not already exist.

You could define the `url` as any destination the node can reach during provisioning - the file does not need to exist on the ZTPServer.

Note: Please see the copy_file documentation for more details.

Install a Specific EOS Image

Objective

I want a specific (v)EOS version to be automatically installed when I provision my node.

Note: This assumes that you've already downloaded the desired (v)EOS image from Arista.

Solution

Let's create a place on the ZTPServer to host some SWIs:

```
# Go to your data_root - by default it's /usr/share/ztpserver
admin@ztpserver:~# cd /usr/share/ztpserver

# Create an images directory
admin@ztpserver:~# mkdir -p files/images

# SCP your SWI into the images directory, name it whatever you like
admin@ztpserver:~# scp admin@otherhost:/tmp/vEOS.swi files/images/vEOS_4.14.5F.swi
```

Now let's create a definition that performs the `install_image` action:

```
# Go to your data_root - by default it's /usr/share/ztpserver
admin@ztpserver:~# cd /usr/share/ztpserver

# Create a definition file
admin@ztpserver:~# vi definitions/tor-definition
```

Add the following lines to your definition, changing values where needed:

```
---
name: static node definition
actions:
  -
    action: install_image
    always_execute: true
    attributes:
      url: files/images/vEOS_4.14.5F.swi
      version: 4.14.5F
    name: "Install 4.14.5F"
```

Note: The definition uses YAML syntax

Explanation

In this case we are hosting the SWI on the ZTPServer, so we just define the `url` in relation to the `data_root`. We could change the `url` to point to another server altogether - the choice is yours. The benefit of hosting the file on the ZTPServer is that we perform an extra checksum step to validate the integrity of the file.

In practice, the node requests its definition during the provisioning process. It sees that it's supposed to perform the `install_image` action, so it requests the `install_image` python script. It then performs an HTTP GET for the `url`. Once it has these locally, it executes the install_image script.

Install a Specific EOS Image without downgrading newer systems

Objective

I want a specific (v)EOS version to be automatically installed when I provision my node but I don't want systems with newer EOS versions to be downgraded

Note: This assumes that you've already downloaded the desired (v)EOS image from Arista.

Solution

Let's create a place on the ZTPServer to host some SWIs:

```
# Go to your data_root - by default it's /usr/share/ztpserver
admin@ztpserver:~# cd /usr/share/ztpserver

# Create an images directory
admin@ztpserver:~# mkdir -p files/images

# SCP your SWI into the images directory, name it whatever you like
admin@ztpserver:~# scp admin@otherhost:/tmp/vEOS.swi files/images/vEOS_4.14.5F.swi
```

Now let's create a definition that performs the `install_image` action:

```
# Go to your data_root - by default it's /usr/share/ztpserver
admin@ztpserver:~# cd /usr/share/ztpserver
```

```
# Create a definition file
admin@ztpserver:~# vi definitions/tor-definition
```

Add the following lines to your definition, changing values where needed. Specifically note the `downgrade: false` attribute.

```
---
name: static node definition
actions:
  -
    action: install_image
    attributes:
      downgrade: false
      url: files/images/vEOS_4.17.1F.swi
      version: 4.17.1F
    name: "Install 4.17.1F"
```

Note: The definition uses YAML syntax

Explanation

The difference between this recipe and the one, above, is setting the `downgrade` attribute to `false`. When downgrades are disabled, an image will only be copied if the running image is older than the image in the ZTP configuration.

Install an Extension

Objective

I want to install an extension on my node automatically.

Solution

Let's create a place on the ZTPServer to host the RPMs:

```
# Go to your data_root - by default it's /usr/share/ztpserver
admin@ztpserver:~# cd /usr/share/ztpserver

# Create an images directory
admin@ztpserver:~# mkdir -p files/rpms

# SCP your SWI into the images directory, name it whatever you like
admin@ztpserver:~# scp admin@otherhost:/tmp/myRPM.rpm files/rpms/myRPM.rpm
```

Now let's create a definition that performs the `install_extension` action:

```
# Go to your data_root - by default it's /usr/share/ztpserver
admin@ztpserver:~# cd /usr/share/ztpserver

# Create a definition file
admin@ztpserver:~# vi definitions/tor-definition
```

Add the following lines to your definition, changing values where needed:

```
---
name: static node definition
actions:
  -
    action: install_extension
    always_execute: true
    attributes:
      url: files/rpms/myRPM.rpm
    name: "Install myRPM extension"
```

Note: The definition uses YAML syntax

Explanation

The `install_extension` will copy the RPM defined in the `url` parameter and copy it to the default extension directory, `/mnt/flash/.extensions`

Note: Please see the install_extension documentation for more details.

Resource Pools

Add a New Resource Pool

Objective

I'd like to add a new resource pool of IP addresses so that I can assign a new IP to each node that gets provisioned.

Note: Resource Pools are simple `key: value` YAML files.

Solution

Create the resource pool

```
# Go to your data_root - by default it's /usr/share/ztpserver
admin@ztpserver:~# cd /usr/share/ztpserver

# Create a resource pool file
admin@ztpserver:~# vi resources/mgmt_ip
```

```
192.168.0.2/24: null
192.168.0.3/24: null
192.168.0.4/24: null
192.168.0.5/24: null
192.168.0.6/24: null
192.168.0.7/24: null
192.168.0.8/24: null
192.168.0.9/24: null
192.168.0.10/24: null
```

Explanation

Resource Pool files are just `key: value` files. The default value for each key should be `null`. This makes the key available for assignment. If you would like to pre-assign a specific node with a particular key, then just put the node's node_id in place of `null`. Resource Pools are analyzed when the `allocate(pool_name)` function is run from a definition. Note that you can also use the `allocate()` function to perform a lookup when a node has already been assigned a key.

Clearing a Resource Pool

Objective

I'd like to reset the values of a resource pool so that all values return to `null`.

Solution

You can use the ztps command line to perform this action.

```
admin@ztpserver:~# ztps --clear-resources
```

Note: This will clear **ALL** resource pools

Explanation

Clearing all resource pools can be done via the command line on the ZTPServer. The command will analyze `data_root/resources` and any file that exists in that directory that resembles a ZTPServer resource pool will be cleared.

Advanced

- *Configuration Management and prep for ZTR*
- *Zero-touch replatement (ZTR)*

Configuration Management and prep for ZTR

Objective

I want to automatically push the startup-config from each node to the corresponding /nodes/ folder whenever changes are made on the node.

Solution

The ZTPServer accepts HTTP PUT requests at `nodes/<node_id>/startup-config`. Therefore, we can configure and event-handler on the node during provisioning which will perform this PUT anytime the startup-config is saved.

1. Create event-handler template

Choose the option that best fits your deployment. The variations are Serial Number or System Mac Address, and Default VRF or Non-Default VRF.

Copy and paste the option text into a new template in:

```
# Go to your data_root - by default it's /usr/share/ztpserver
admin@ztpserver:~# cd /usr/share/ztpserver

# Make sure you have a directory for templates
admin@ztpserver:~# mkdir -p files/templates

# Create a static config block
admin@ztpserver:~# vi files/templates/config-push.template
```

Note: Notice the `$ztpserver`, `$port` and `$vrf_name` variables. You can hardcode these in the template or abstract these to the definition or attributes file (as shown in the next recipe).

Option 1: Using SystemMac and Default VRF

```
event-handler configpush
 trigger on-startup-config
 action bash export SYSMAC=`FastCli -p 15 -c 'show ver | grep MAC | cut -d" " -f 5' |␣
↪sed 's/[.]*//g'`; curl http://$ztpserver:$port/nodes/$SYSMAC/startup-config -H
↪"content-type: text/plain" --data-binary @/mnt/flash/startup-config -X PUT
```

Option 2: Using SystemMac and Non-Default VRF

```
event-handler configpush
 trigger on-startup-config
 ! For non-default VRF, use:
 action bash export SYSMAC=`FastCli -p 15 -c 'show ver | grep MAC | cut -d" " -f 5' |␣
↪sed 's/[.]*//g'`; sudo ip netns exec ns-$vrf_name curl http://$ztpserver:$port/
↪nodes/$SYSMAC/startup-config -H "content-type: text/plain" --data-binary @/mnt/
↪flash/startup-config -X PUT
```

Option 3: Using Serial Number and Default VRF

```
event-handler configpush
 trigger on-startup-config
```

```
 ! For serial number, default VRF:
 action bash export SERIAL=`FastCli -p 15 -c 'show ver' | grep Serial | tr -s ' ' |␣
↪cut -d ' ' -f 3 | tr -d '\r'`; curl http://$ztpserver:$port/nodes/$SERIAL/startup-
↪config -H "content-type: text/plain" --data-binary @/mnt/flash/startup-config -X PUT
```

Option 4: Using Serial Number and Non-Default VRF

```
event-handler configpush
 trigger on-startup-config
 ! For serial number, non-default VRF:
 action bash export SERIAL=`FastCli -p 15 -c 'show ver' | grep Serial | tr -s ' ' |␣
↪cut -d ' ' -f 3 | tr -d '\r'`; sudo ip netns exec ns-$vrf_name curl http://
↪$ztpserver:$port/nodes/$SERIAL/startup-config -H "content-type: text/plain" --data-
↪binary @/mnt/flash/startup-config -X PUT
```

Zero-touch replatement (ZTR)

Objective

I replaced a switch with a new one and want it to provision with the same configuration and, optionally, EOS version as the node it replaced.

Solution

ZTPServer first looks for a pre-existing definition for a node in the `<configdir>/nodes/<node-id>` directory before trying to match through neighbordb, etc. Thus, you can make ZTPServer think it has already seen this node by renaming, linking or copying the old-node's directory to the new-node's unique-id before powering the switch on for the first time.

Moving (renaming) or linking are most commonly used, however, making a recursive copy will ensure that the last-known configuration of the previous node remains stored as a backup.

```
cd /usr/share/ztpserver/nodes
ln -s <old-node_id> <new-node_id>
```

Puppet Agent - Bootstrap EOS

Bootstrap EOS to Puppet

Objective

I want to bootstrap an EOS node with the Puppet agent.

Solution

Note: Prior to EOS 4.14.5, eAPI must be configured with HTTPS or HTTP and a flash:eapi.conf must be created for rbeapi. Starting with EOS 4.14.5, rbeapi can use unix-sockets to communicate with eAPI, locally.

Download the Puppet Enterprise agent (may be used with Puppet Enterprise or Open Source) from PuppetLabs and the Ruby client for eAPI (pe-rbeapi) SWIX from GitHub. Place these files in /usr/share/ztpserver/files/puppet/‘‘

```
---
name: puppet-test
actions:
  -
    name: "Install Puppet agent"
    action: install_extension
    always_execute: true
    attributes:
      url: files/puppet/puppet-enterprise-3.8.2-eos-4-i386.swix
  -
    name: "Install rbeapi - Ruby client for eAPI"
    action: install_extension
    always_execute: true
    attributes:
      url: files/puppet/rbeapi-0.3.0.swix
  -
    name: "Configure host alias and eAPI for Puppet"
    action: add_config
    attributes:
      url: files/templates/puppet.template
      variables:
        hostname: allocate('mgmt_hostnames')
        domainname: example.com
        puppetmaster: 172.16.130.10
        ntpserver: 66.175.209.17
    onstart: "Starting to configure EOS for Puppet"
    onsuccess: "SUCCESS: Base config for Puppet"
```

```
!
alias puppet bash sudo /opt/puppet/bin/puppet
!
hostname $hostname
!
ip domain-name $domainname
!
ip host puppet $puppetmaster
!
ntp server $ntpserver prefer iburst
!
management api http-commands
   no protocol https
   protocol unix-socket
   no shutdown
!
```

Explanation

Here we use the `install_extension` action to install the Puppet agent and Ruby client for eAPI, then apply a minimal configuration so the Puppet agent can generate its SSL keys and contact the Puppet Master. The attributes listed in the `add_config` action will be passed to the node so that it is able to properly generate its SSL keypair and certificate signing request (CSR) and validate the Puppet master's certificate.

Note: For more Action recipes see the Actions section.

Ansible - Bootstrap EOS

Introduction

The following recipes will help you bootstrap Arista EOS switches for use with Ansible. Please review the Ansible-EOS documentation to determine your preferred connection type: SSH or eAPI.

Note: Please contact us if you are interested in dynamically adding your nodes to Ansible Tower. We have various examples that utilize the Tower API to add your node to a specific Tower inventory and/or group.

Bootstrap EOS for Ansible using SSH

Objective

I want to bootstrap an EOS node so that I can use Ansible to SSH to the node.

Solution

Note: Prior to EOS 4.14.5, eAPI must be configured with HTTPS or HTTP and a flash:eapi.conf must be created for pyeapi or the eAPI credentials must be passed in the Ansible task using meta arguments. Starting with EOS 4.14.5, pyeapi can use unix-sockets to communicate with eAPI, locally.

Step 1 Gather Ansible Control Host SSH Key

First, store the Ansible Control Host SSH key on the ZTPServer (or make it available via URL). When the `configure_ansible_client` action runs it will create a bash user on the switch and put this key in `~/.ssh/authorized_keys`.

In `[DATA_ROOT]/files/ssh/key.pub`

```
ssh-rsa AAAAB3NzaC1yc....rest of public key......
```

Step 2 Create a management IP resource pool

Reference this recipe for an example.

Step 3 Create eAPI configuration

In `[DATA_ROOT]/files/templates/eapi.template`

Option A Using Unix Sockets (4.14.5+)

```
!
management api http-commands
   no protocol https
   protocol unix-socket
   no shutdown
!
```

Option B Using HTTPS

```
!
management api http-commands
   no shutdown
!
```

Option C Using HTTP

```
!
management api http-commands
   no shutdown
   no protocol https
   protocol http
!
```

Step 4 Create a definition

Let's use the `configure_ansible_client` action to create the desired SSH user.

```
---
actions:
-
  action: configure_ansible_client
  attributes:
    key: files/ssh/key.pub
    user: ansible
    passwd: password
    group: eosadmin
    root: "/persist/local/"
  name: "Configure Ansible"
-
  action: add_config
  attributes:
    url: files/templates/ma1.template
    variables:
      ipaddress: allocate('mgmt_subnet')
  name: "configure ma1"
-
  action: add_config
  attributes:
```

```
    url: files/templates/eapi.template
  name: "Enable eAPI"
```

Explanation

Here we use the `add_config` action to load the switch with a standard eAPI configuration as well as assign Management1 interface an IP address allocated from the mgmt_subnet pool. Note that ZTPServer supports custom allocate scripts that could dynamically assign an IP address from your own IPAM. Also, the `configure_ansible_client` action is called. This client-side action will create a bash user, with the specified name, and install any SSH keys provided to `~/.ssh/authorized_keys`. This is helpful because it takes care of authentication between the Ansible Control host and the switch. The action also writes to `rc.eos` to create this user on every boot (since it would normally be blown away).

Bootstrap EOS for Ansible using eAPI

Objective

I want to bootstrap an EOS node so that I can use Ansible in connection:local mode and connect to my switch via eAPI.

Solution

Step 1 Create a management IP resource pool

Reference this recipe for an example.

Step 2 Create eAPI configuration

In `[DATA_ROOT]/files/templates/eapi.template`

Option A Using HTTPS

```
!
management api http-commands
   no shutdown
!
```

Option B Using HTTP

```
!
management api http-commands
   no shutdown
   no protocol https
   protocol http
!
```

Step 3 Create a definition

```
---
actions:
-
  action: add_config
  attributes:
    url: files/templates/ma1.template
```

```
    variables:
      ipaddress: allocate('mgmt_subnet')
  name: "configure ma1"
-
  action: add_config
  attributes:
    url: files/templates/eapi.template
  name: "Enable eAPI"
```

Explanation

Here we use the `add_config` action to load the switch with a standard eAPI configuration as well as assign Management1 interface an IP address allocated from the mgmt_subnet pool. Note that ZTPServer supports custom allocate scripts that could dynamically assign an IP address from your own IPAM.

Note: For more Action recipes see the Actions section.

Run ZTPServer as a VM on EOS

Introduction

Bootstrapping network devices, much like bootstrapping servers, requires a server in place to handle that function. Often, it is cumbersome to have that server ready before the network is up and running. Therefore, it will be very convenient to have a server up and running, along with the first node in the network fabric, to handle bootstrapping for the rest of the infrastructure.

Arista EOS provides the capability to run VMs on top of EOS, therefore making the above scenario possible. The following set of recipes will help you perform the necessary steps to streamline your data center network bootstrapping process:

- You can have everything prepared and stored on a USB key.
- Plug in the USB key to the first SPINE switch in the data center.
- The rest of the data center fabric will be bootstrapped automatically!

There are 3 different deployment topologies, and your network design should fall into one of them. Each topology requires slightly different recipes, and they are explained in the following sections.

- **L2L3WM** : a L2 MLAG or L3 ECMP fabric with an out-of-band management network (switches managed via the management port)
- **L2WOM** : a L2 MLAG fabric without an out-of-band management network (switches managed in-band via SVI)
- **L3WOM** : a L3 ECMP fabric without an out-of-band management network (switches managed in-band via loopback)

ZTPServer VM on EOS in a L2L3WM

Files Needed

- `ztps.vmdk` : the VM disk image for the ZTPServer VM

- `startup-config`: a text file (with no extension)
- `ztps.sh` : a bash shell script
- `ztps.xml` : an xml file
- `fullrecover` : an empty text file (with no extension)
- `boot-config` : a text file (with no extension); contains a single line: `SWI=flash:EOS.swi`
- `EOS.swi` : download an EOS image and rename it to `EOS.swi`

ztps.vmdk

Objective

I want to create a ZTPServer vmdk file to use on EOS.

Solution

The ZTPServer vmdk file can be created using either methods below:

1. Automatically Create a Full-Featured ZTPServer: https://github.com/arista-eosplus/packer-ZTPServer
2. Create your own VM and install ZTPServer as intructed in the "Installation" section

Explanation

The turnkey solution detailed on the github will create a full featured ztps.vmdk by executing a single command. The vmdk created using this method comes with certain parameters pre-defined (i.e. domain-name, root user credential, IP address, etc). If desired, you can change these parameters by logging into the VM after it's created.

The second method requires more manual work compare to the first method, but may be more suitable if you already have a VM build to your needs and simply want to add ZTPServer to it.

startup-config

Objective

I need to prepare a startup-config for the first SPINE switch to enable ZTPServer.

Solution

Essential parts of the configuration:

- `event-handler ztps` : used to start the shell script `ztps.sh`
- `virtual-machine ztps` : used to start the ZTPServer VM on EOS

```
interface Management1
  ip address 192.168.1.10/24

event-handler ztps
  trigger on-boot
```

```
  action bash /mnt/flash/ztps.sh &
  delay 300

virtual-machine ztps
  config-file flash:/ztps.xml
  enable
```

Explanation

The `event-handler ztps` is triggered on-boot to kickstart the shell script `ztps.sh`. There is a delay of 300 seconds before the script will be executed, to make sure all the necessary systems are in place before we run the script. For details of the script please see the `ztps.sh` section.

External systems will connect to the VM via the management network. The host switch will connect to the VM via the Linux bridge (See `ztps.sh`). Therefore in this scenario we will need to have 2 interfaces on the ZTPServer VM.

For details of the shell script `ztps.sh` please refer to the corresponding sectio below.

ztps.sh

Objective

I want to create a shell script to set up all the necessary environment for ZTPServer when the switch boots up.

Solution

```
#!/bin/bash
# This script is used with the event-handler so that on-boot, we will create linux␣
↪bridge,
#enable ip.forwarding, restart the ZTPS VM, and start DHCPD
logger -t "ZTPS" -p local0.info "Starting the process for ZTPS VM deployment"

# Create Linux Bridge
sudo brctl addbr br0
sudo ifconfig br0 up
sudo ifconfig br0 172.16.130.254/24

logger -t "ZTPS" -p local0.info "Linux Bridge created"

#Now lets restart the  ZTPS VM
sudo echo -e "enable\nconfigure terminal\nvirtual-machine ztps restart\n" | FastCli -
↪M -e -p 15

logger -t "ZTPS" -p local0.info "ZTPS VM restarted"
```

Explanation

In order to enable connectivity to the VM locally (from the host switch), a Linux bridge interface needs to be created and assigned an IP in the same subnet as one of the interfaces on the VM.

The ZTPServer VM needs to be restarted after the switch boots up.

Note: The ZTPServer VM needs to have its default gateway pointed to the default gateway of the management network.

ztps.xml

Objective

I want to prepare a KVM custom xml file to enable a VM on EOS.

Solution

Key parts of the xml file to pay attention to:

- `<domain type='kvm' id='1'>` : id needs to be unique (if more than 1 VM)
- `<driver name='qemu' type='vmdk'/>` : make sure the type is `vmdk`
- `<source file='/mnt/usb1/ztps.vmdk'/>`: make sure the path is correct
- **Interface definition section** :
  - MAC address in the xml need to match the MAC address of the interfaces on the ZTPServer VM.
  - The first interface type is direct and is mapped to ma1. This is the interface that will be used for other switches to reach the VM.
  - The second interface type is bridge and is using Linux bridge. This interface is solely used for local host switch to VM connectivity.

```
<domain type='kvm' id='1'>
  <name>ztps</name>
  <memory>1048576</memory>
  <currentMemory>1048576</currentMemory>
  <vcpu>1</vcpu>
  <os>
    <type arch='x86_64' machine='pc-i440fx-1.4'>hvm</type>
    <boot dev='hd'/>
  </os>
  <features>
    <acpi/>
    <apic/>
    <pae/>
  </features>
  <clock offset='utc'/>
  <on_poweroff>destroy</on_poweroff>
  <on_reboot>restart</on_reboot>
  <on_crash>restart</on_crash>
  <devices>
    <emulator>/usr/bin/qemu-system-x86_64</emulator>
    <disk type='file' device='disk'>
      <driver name='qemu' type='vmdk'/>
      <source file='/mnt/usb1/ztps.vmdk'/>
      <target dev='hda' bus='ide'/>
      <alias name='ide0-0-0'/>
      <address type='drive' controller='0' bus='0' unit='0'/>
```

```
    </disk>
    <controller type='ide' index='0'>
      <alias name='ide0'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x01' function='0x1'/>
    </controller>
    <interface type='direct'>
      <mac address='08:00:27:bc:d7:38'/>
      <source dev='ma1' mode='bridge'/>
      <target dev='macvtap0'/>
      <model type='e1000'/>
      <alias name='net0'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x03' function='0x0'/>
    </interface>
    <interface type='bridge'>
      <mac address='08:00:27:85:0c:f8'/>
      <source bridge='br0'/>
      <target dev='macvtap1'/>
      <model type='e1000'/>
      <alias name='net1'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x04' function='0x0'/>
    </interface>
    <serial type='pty'>
      <source path='/dev/pts/5'/>
      <target port='0'/>
      <alias name='serial0'/>
    </serial>
    <console type='pty' tty='/dev/pts/5'>
      <source path='/dev/pts/5'/>
      <target type='serial' port='0'/>
      <alias name='serial0'/>
    </console>
    <input type='tablet' bus='usb'>
      <alias name='input0'/>
    </input>
    <input type='mouse' bus='ps2'/>
    <graphics type='vnc' port='5900' autoport='no' listen='0.0.0.0'/>
    <video>
      <model type='vga' vram='8192' heads='1'/>
      <alias name='video0'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x02' function='0x0'/>
    </video>
    <memballoon model='virtio'>
      <alias name='balloon0'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x05' function='0x0'/>
    </memballoon>
  </devices>
</domain>
```

Explanation

The interface definition section defines how the interface(s) of the VM should be initialized. Since the vmdk already has interfaces defined/initialized, we have to use the same MAC address in the KVM definition file.

In the first interface definition we use `interface type='direct'`. In this configuration we map the first interface of the VM to the `ma1` interface directly, enabling connectivity to the VM from external of the host switch. However, `interface type='direct'` does not allow for host switch to VM connectivity, therefore we need to

define a second interface with `interface type='bridge'` and map that to the Linux bridge for this puspose.

The reason we could not just bridge ma1 with the Linux bridge (and therefore just use one interface to enable both local and external connectivity) is because when we enslave an interface to br0, that interface cannot have an IP address on it, otherwise the connectivity would break.

ZTPServer VM on EOS in a L2WOM

Files Needed

- `ztps.vmdk` : the VM disk image for the ZTPServer VM
- `startup-config`: a text file (with no extension)
- `ztps.sh` : a bash shell script
- `ztps.xml` : an xml file
- `fullrecover` : an empty text file (with no extension)
- `boot-config` : a text file (with no extension); contains a single line: `SWI=flash:EOS.swi`
- `EOS.swi` : download an EOS image and rename it to `EOS.swi`

ztps.vmdk

Objective

I want to create a ZTPServer vmdk file to use on EOS.

Solution

The ZTPServer vmdk file can be created using either methods below:

1. Automatically Create a Full-Featured ZTPServer: https://github.com/arista-eosplus/packer-ZTPServer
2. Create your own VM and install ZTPServer as intructed in the "Installation" section

Explanation

The turnkey solution detailed on the github will create a full featured ztps.vmdk by executing a single command. The vmdk created using this method comes with certain parameters pre-defined (i.e. domain-name, root user credential, IP address, etc). If desired, you can change these parameters by logging into the VM after it's created.

The second method requires more manual work compare to the first method, but may be more suitable if you already have a VM build to your needs and simply want to add ZTPServer to it.

startup-config

Objective

I need to prepare a startup-config for the first SPINE switch to enable ZTPServer.

Solution

Essential parts of the configuration:

- `event-handler ztps` : used to start the shell script `ztps.sh`
- `virtual-machine ztps` : used to start the ZTPServer VM on EOS

```
interface Vlan1
  ip address 192.168.1.10/24

event-handler ztps
  trigger on-boot
  action bash /mnt/flash/ztps.sh &
  delay 300

virtual-machine ztps
  config-file flash:/ztps.xml
  enable
```

Explanation

The `event-handler ztps` is triggered on-boot to kickstart the shell script `ztps.sh`. There is a delay of 300 seconds before the script will be executed, to make sure all the necessary systems are in place before we run the script. For details of the script please see the `ztps.sh` section.

External systems will connect to the VM via Vlan1 (other VLANs can be used as well). The host switch will connect to the VM via the Linux bridge (See `ztps.sh`). Therefore in this scenario we will need to have 2 interfaces on the ZTPServer VM.

For details of the shell script `ztps.sh` please refer to the corresponding section below.

ztps.sh

Objective

I want to create a shell script to set up all the necessary environment for ZTPServer when the switch boots up.

Solution

```
#!/bin/bash
# This script is used with the event-handler so that on-boot, we will create linux␣
↪bridge,
#enable ip.forwarding, restart the ZTPS VM, and start DHCPD
logger -t "ZTPS" -p local0.info "Starting the process for ZTPS VM deployment"

# Create Linux Bridge
sudo brctl addbr br0
sudo ifconfig br0 up
sudo ifconfig br0 172.16.130.254/24

logger -t "ZTPS" -p local0.info "Linux Bridge created"

#Now lets restart the  ZTPS VM
```

```
sudo echo -e "enable\nconfigure terminal\nvirtual-machine ztps restart\n" | FastCli -
↪M -e -p 15

logger -t "ZTPS" -p local0.info "ZTPS VM restarted"
```

Explanation

In order to enable connectivity to the VM locally (from the host switch), a Linux bridge interface needs to be created and assigned an IP in the same subnet as one of the interfaces on the VM.

The ZTPServer VM needs to be restarted after the switch boots up.

Note: The ZTPServer VM needs to have its default gateway pointed to the default gateway of Vlan1 (or your choice of VLAN).

---

ztps.xml

Objective

I want to prepare a KVM custom xml file to enable a VM on EOS.

Solution

Key parts of the xml file to pay attention to:

- `<domain type='kvm' id='1'>` : id needs to be unique (if more than 1 VM)
- `<driver name='qemu' type='vmdk'/>` : make sure the type is `vmdk`
- `<source file='/mnt/usb1/ztps.vmdk'/>`: make sure the path is correct
- **Interface definition section** :
  - MAC address in the xml need to match the MAC address of the interfaces on the ZTPServer VM.
  - The first interface type is direct and is mapped to vlan1. This is the interface that will be used for other switches to reach the VM.
  - The second interface type is bridge and is using Linux bridge. This interface is solely used for local host switch to VM connectivity.

```
<domain type='kvm' id='1'>
  <name>ztps</name>
  <memory>1048576</memory>
  <currentMemory>1048576</currentMemory>
  <vcpu>1</vcpu>
  <os>
    <type arch='x86_64' machine='pc-i440fx-1.4'>hvm</type>
    <boot dev='hd'/>
  </os>
  <features>
    <acpi/>
    <apic/>
```

```
    <pae/>
  </features>
  <clock offset='utc'/>
  <on_poweroff>destroy</on_poweroff>
  <on_reboot>restart</on_reboot>
  <on_crash>restart</on_crash>
  <devices>
    <emulator>/usr/bin/qemu-system-x86_64</emulator>
    <disk type='file' device='disk'>
      <driver name='qemu' type='vmdk'/>
      <source file='/mnt/usb1/ztps.vmdk'/>
      <target dev='hda' bus='ide'/>
      <alias name='ide0-0-0'/>
      <address type='drive' controller='0' bus='0' unit='0'/>
    </disk>
    <controller type='ide' index='0'>
      <alias name='ide0'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x01' function='0x1'/>
    </controller>
    <interface type='direct'>
      <mac address='08:00:27:bc:d7:38'/>
      <source dev='vlan1' mode='bridge'/>
      <target dev='macvtap0'/>
      <model type='e1000'/>
      <alias name='net0'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x03' function='0x0'/>
    </interface>
    <interface type='bridge'>
      <mac address='08:00:27:85:0c:f8'/>
      <source bridge='br0'/>
      <target dev='macvtap1'/>
      <model type='e1000'/>
      <alias name='net1'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x04' function='0x0'/>
    </interface>
    <serial type='pty'>
      <source path='/dev/pts/5'/>
      <target port='0'/>
      <alias name='serial0'/>
    </serial>
    <console type='pty' tty='/dev/pts/5'>
      <source path='/dev/pts/5'/>
      <target type='serial' port='0'/>
      <alias name='serial0'/>
    </console>
    <input type='tablet' bus='usb'>
      <alias name='input0'/>
    </input>
    <input type='mouse' bus='ps2'/>
    <graphics type='vnc' port='5900' autoport='no' listen='0.0.0.0'/>
    <video>
      <model type='vga' vram='8192' heads='1'/>
      <alias name='video0'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x02' function='0x0'/>
    </video>
    <memballoon model='virtio'>
      <alias name='balloon0'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x05' function='0x0'/>
```

```
    </memballoon>
  </devices>
</domain>
```

Explanation

The interface definition section defines how the interface(s) of the VM should be initialized. Since the vmdk already has interfaces defined/initialized, we have to use the same MAC address in the KVM definition file.

In the first interface definition we use `interface type='direct'`. In this configuration we map the first interface of the VM to the `vlan1` interface directly, enabling connectivity to the VM from external of the host switch. However, `interface type='direct'` does not allow for host switch to VM connectivity, therefore we need to define a second interface with `interface type='bridge'` and map that to the Linux bridge for this puspose.

The reason we could not just bridge Vlan1 with the Linux bridge (and therefore just use one interface to enable both local and external connectivity) is because when we enslave an interface to br0, that interface cannot have an IP address on it, otherwise the connectivity would break.

ZTPServer VM on EOS in a L3WOM

Files Needed

- `ztps.vmdk` : the VM disk image for the ZTPServer VM
- `startup-config`: a text file (with no extension)
- `ztps.sh` : a bash shell script
- `ztps.xml` : an xml file
- `dhcpd.conf` : a text file for Linux dhcpd configuration
- `dhcpd.rpm` : a DHCP server RPM to be installed on EOS
- `ztps_daemon` : a python script
- `fullrecover` : an empty text file (with no extension)
- `boot-config` : a text file (with no extension); contains a single line: `SWI=flash:EOS.swi`
- `boot-extention`: a text file (with no extention); contains a single like: `dhcpd.rpm`
- `EOS.swi` : download an EOS image and rename it to `EOS.swi`

ztps.vmdk

Objective

I want to create a ZTPServer vmdk file to use on EOS.

Solution

The ZTPServer vmdk file can be created using either methods below:

1. Automatically Create a Full-Featured ZTPServer: https://github.com/arista-eosplus/packer-ZTPServer

2. Create your own VM and install ZTPServer as intructed in the "Installation" section

Explanation

The turnkey solution detailed on the github will create a full featured ztps.vmdk by executing a single command. The vmdk created using this method comes with certain parameters pre-defined (i.e. domain-name, root user credential, IP address, etc). If desired, you can change these parameters by logging into the VM after it's created.

The second method requires more manual work compare to the first method, but may be more suitable if you already have a VM build to your needs and simply want to add ZTPServer to it.

startup-config

Objective

I need to prepare a startup-config for the first SPINE switch to enable ZTPServer.

Solution

Essential parts of the configuration:

- `interface Loopback2` : need a loopback interface on the same subnet as the VM
- `daemon ztps` : used to run the `ztps.daemon` python script in the background
- `event-handler ztps` : used to start the shell script `ztps.sh`
- `virtual-machine ztps` : used to start the ZTPServer VM on EOS
- `management api http-commands`: need to enable eAPI for `daemon ztps` to function

```
interface Loopback2
  ip address 172.16.130.253/24

daemon ztps
  command /mnt/flash/ztps_daemon &

event-handler ztps
  trigger on-boot
  action bash /mnt/flash/ztps.sh &
  delay 300

virtual-machine ztps
  config-file flash:/ztps.xml
  enable

management api http-commands
  protocol http localhost
  no shutdown
```

Explanation

The `event-handler ztps` is triggered on-boot to kickstart the shell script `ztps.sh`. There is a delay of 300 seconds before the script will be executed, to make sure all the necessary systems are in place before we run the script.

For details of the script please see the `ztps.sh` section.

The `management api http-commands` section enables Arista eAPI on the host swithc; eAPI is leveraged by the `ztps_daemon`. eAPI can be accessed remotely via http or https, or it can be accessed locally via http, or by binding to a UNIX socket (only available on 4.14.5F onward). Since the daemon is a script that runs locally, we can either enalbe eAPI on the localhost via http (if you are running 4.14.5F or later), or we can just enable eAPI over https (this will require authentication).

The `daemon ztps` section runs a python script in the back ground as a daemon to restart DHCPD whenever an interface comes up.

For details of the shell script `ztps.sh` and the python script `ztps_daemon` please refer to the corresponding sectio below.

Note: The loopback interface is only needed if you plan to bootstrap a L3 ECMP fabric without a management network. In this scenario, the loopback address needs to be advertised in the ECMP routing protocol to enable connectivity for the downstream deviecs in the fabric.

---

ztps.sh

Objective

I want to create a shell script to set up all the necessary environment for ZTPServer when the switch boots up.

Solution

```
#!/bin/bash
# This script is used with the event-handler so that on-boot, we will create linux␣
↪bridge,
#enable ip.forwarding, restart the ZTPS VM, and start DHCPD
logger -t "ZTPS" -p local0.info "Starting the process for ZTPS VM deployment"

# Create Linux Bridge
sudo brctl addbr br0
sudo ifconfig br0 up
sudo ifconfig br0 172.16.130.254/24

logger -t "ZTPS" -p local0.info "Linux Bridge created"

# Enable ip.forwarding
sudo sysctl net.ipv4.conf.all.forwarding=1
sudo sysctl net.ipv4.ip_forward=1

logger -t "ZTPS" -p local0.info "ip.forwarding enabled"

# Move the DHCP server RPM to the appropriate folder on EOS for installation
# Move the dhcpd.conf file to the appropriate folder
sudo cp /mnt/flash/dhcp-4.2.0-23.P2.fc14.i686.rpm /mnt/flash/.extensions/dhcpd.rpm
sudo cp /mnt/flash/dhcpd.conf /etc/dhcp/
sudo /usr/sbin/dhcpd
sleep 5

#make sure dhcpd is running before we continue
```

```
ps aux | grep "dhcpd" | grep -v grep
if [ $? -eq 0 ]
then
{
logger -t "ZTPS" -p local0.info "DHCPD is running. Restart ZTPS VM."

#Now lets restart the  ZTPS VM
sudo echo -e "enable\nconfigure terminal\nvirtual-machine ztps restart\n" | FastCli -
↪M -e -p 15

logger -t "ZTPS" -p local0.info "ZTPS VM restarted"

exit 0
}
else
  logger -t "ZTPS" -p local0.info "Looks like DHCPD didn't start. Lets sleep for a␣
↪few seconds and try again"
  sleep 10
fi
```

Explanation

In order to enable connectivity to the VM from both remotely and locally (from the host switch), a Linux bridge interface needs to be created and assigned an IP in the same subnet as the VM; Linux `ip.forwarding` also needs to be enabled in the kernel for the packets to be routed to the VM.

EOS does not come with dhcpd preinstalled, there a DHCP-Server RPM needs to be downloaded, installed and started. Dowdload the RPM from here and rename it to `dhcpd.rpm`. The RPM needs to be moved to the `/mnt/flash/.extension` location, and a `boot-extension` file, with the RPM specified, needs to be present in `/mnt/flash` in order for the RPM to be installed persistently after a reboot.

The ZTPServer VM needs to be restarted after the switch boots up.

Note: The ZTPServer VM needs to have its default gateway pointed to the br0 interface IP address.

---

ztps.xml

Objective

I want to prepare a KVM custom xml file to enable a VM on EOS.

Solution

Key parts of the xml file to pay attention to:

- `<domain type='kvm' id='1'>` : in case multiple VMs are running on the system, make sure the configured ID is unique
- `<driver name='qemu' type='vmdk'/>` : make sure the type is `vmdk`
- `<source file='/mnt/usb1/ztps.vmdk'/>`: make sure the path is correct

- `<mac address='08:00:27:85:0c:f8'/>` : make sure this MAC matches the MAC address of the interface on the ZTPServer VM that you intend to use for connectivity
- `<target dev='vnet0'/>` : make sure the target device type is `vnet0`

```
<domain type='kvm' id='1'>
  <name>ztps</name>
  <memory>1048576</memory>
  <currentMemory>1048576</currentMemory>
  <vcpu>1</vcpu>
  <os>
    <type arch='x86_64' machine='pc-i440fx-1.4'>hvm</type>
    <boot dev='hd'/>
  </os>
  <features>
    <acpi/>
    <apic/>
    <pae/>
  </features>
  <clock offset='utc'/>
  <on_poweroff>destroy</on_poweroff>
  <on_reboot>restart</on_reboot>
  <on_crash>restart</on_crash>
  <devices>
    <emulator>/usr/bin/qemu-system-x86_64</emulator>
    <disk type='file' device='disk'>
      <driver name='qemu' type='vmdk'/>
      <source file='/mnt/usb1/ztps.vmdk'/>
      <target dev='hda' bus='ide'/>
      <alias name='ide0-0-0'/>
      <address type='drive' controller='0' bus='0' unit='0'/>
    </disk>
    <controller type='ide' index='0'>
      <alias name='ide0'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x01' function='0x1'/>
    </controller>
    <interface type='bridge'>
      <mac address='08:00:27:85:0c:f8'/>
      <source bridge='br0'/>
      <target dev='vnet0'/>
      <model type='e1000'/>
      <alias name='net0'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x04' function='0x0'/>
    </interface>
    <serial type='pty'>
      <source path='/dev/pts/5'/>
      <target port='0'/>
      <alias name='serial0'/>
    </serial>
    <console type='pty' tty='/dev/pts/5'>
      <source path='/dev/pts/5'/>
      <target type='serial' port='0'/>
      <alias name='serial0'/>
    </console>
    <input type='tablet' bus='usb'>
      <alias name='input0'/>
    </input>
    <input type='mouse' bus='ps2'/>
    <graphics type='vnc' port='5900' autoport='no' listen='0.0.0.0'/>
```

```
    <video>
      <model type='vga' vram='8192' heads='1'/>
      <alias name='video0'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x02' function='0x0'/>
    </video>
    <memballoon model='virtio'>
      <alias name='balloon0'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x05' function='0x0'/>
    </memballoon>
  </devices>
</domain>
```

Explanation

The interface definition section defines how the interface(s) of the VM should be initialized. Since the vmdk already has interfaces defined/initialized, we have to use the same MAC address in the KVM definition file.

The target device type should be vnet0 to enable connectivity to the VM from both remotely and locally from the host switch. Another choice here is the macvtap device type but this type prohibits connectivity for any locally routed packets (i.e. when the routing action to the VM takes place on the host switch).

dhcpd.conf

Objective

I want to prepare a dhcpd.conf file for running DHCPD on EOS.

Solution

```
class "ARISTA" {
  match if substring(option vendor-class-identifier, 0, 6) = "Arista";
  option bootfile-name "http://172.16.130.10:8080/bootstrap";
}

# Example
subnet 10.1.1.0 netmask 255.255.255.252 {
  option routers 10.1.1.1;
  default-lease-time 86400;
  max-lease-time 86400;
  pool {
        range 10.1.1.2 10.1.1.2;
        allow members of "ARISTA";
  }
}
```

Explanation

The `class "ARISTA"` section defines a match criteria so that any subnet defition that uses this class would only allocate IPs if the requestor is an Arista device. This class also defines a bootstrap file that will be downloaded to the requestor.

Note: The IP address and TCP port number defined for the bootfile needs to match the ZTPServer VM configuration.

The subnet section provides an example to show you how it can be defined. If you are bootstrapping a L3 ECMP network without a management network, this section needs to be repeated for every p-to-p links connecting to every leaf switches.

Note: The ZTPServer VM also runs dhcpd, but in the scenario of L3 ECMP without a management network, we are unable to leverage that. This is because DHCP relay from the host switch to the VM is currently not supported in EOS.

ztps_daemon

Objective

I want to create a python script that restarts DHCPD whenever an interface comes up.

Solution

```
#!/usr/bin/env python

import jsonrpclib
import os
import time

#PROTO = "https"
#USERNAME = "admin"
#PASSWORD = "admin"
#HOSTNAME = "172.16.130.20"

class EapiClient(object):
    '''
    Instantiate a Eapi connection client object
    for interacting with EAPI
   '''

  def __init__(self):
    # For EOS 4.14.5F and later, you can enable locally run scripts without needing␣
↪to authenticate
    # If you are running earlier versions, just uncomment next line and also the␣
↪CONSTANTS above
    #switch_url = '{}://{}:{}@{}/command-api'.format(PROTO, USERNAME, PASSWORD,␣
↪HOSTNAME)
    switch_url = 'http://localhost:8080/command-api'
    self.client = jsonrpclib.Server(switch_url)

  def connected_interfaces(self):
    cmd = "show interfaces status connected"
    response = self.client.runCmds(1, [cmd])[0]
    connected_intfs = response['interfaceStatuses'].keys()
    return connected_intfs

def restart_dhcpd(eapi):
```

```
    '''
    Monitor the connected interfaces.
    If there are newly connected interface(s), restart dhcpd
    '''
    connected_intfs = []

    while True:
        new_connected_intfs = eapi.connected_interfaces()
        for intf in new_connected_intfs:
            if intf not in connected_intfs:
                os.system('sudo service dhcpd restart')

    connected_intfs = new_connected_intfs
    time.sleep(10)

def main():
  eapi = EapiClient()
  restart_dhcpd(eapi)

if __name__ == '__main__':
  try:
    main()
  except KeyboardInterrupt:
    pass
```

Explanation

DHCPD only binds to interfaces that are UP when the process started. Since we are running DHCPD directly on the SPINE switch, there is no gaurantee that the interfaces connected to the LEAFs are up when DHCPD started. Therefore, we need to run a script/daemon in the background to continuously check the connected interface status, and if new interfaces came up, DHCPD would be restarted to bind to the newly connected interfaces.

Deployment Steps

Objective

I want to use a single USB key to bootstrap my entire data center fabric.

Solution

Follow the steps below:

1. Obtain an USB key that's at least 4GB and format it with either MS-DOS or FAT file system
2. Copy all the files listed in the "Files Needed" section onto the USB key
3. Plug the USB key into the USB port on the first SPINE switch
4. Sit back and watch your data center network fabric bring itself up!

Note: All files and directories present on the USB flash drive will be copied to the switch. It is recommended that the USB drive contains only the three files listed above.

Explanation

The USB key method leverages the Arista Password Recovery mechanism. When the `fullrecover` and `boot-config` file is present on the USB key, the system will check the timestamp on the `boot-config` file.If the timestamp is different, all files on the USB key will be copied to the flash on the switch, and the switch will be rebooted and come up with the `startup-config` and the `EOS.swi` included on the USB key.

Tips and tricks

How do I update my local copy of ZTPServer from GitHub?

Automatically

Go to the ZTPServer directory where you previously cloned the GitHub repository and execute: `./utils/refresh_ztps [-b <branch>] [-f <path>]`

- <branch> can be any branch name in the Git repo. Typically this will be one of:
  - "master" - for the latest release version
  - "vX.Y.Z-rc" - for beta testing the next X.Y.Z release-candidate
  - "develop" (DEFAULT) - for the latest bleeding-edge development branch
- <path> is the base directory of the ztpserver installation.
  - /usr/share/ztpserver (DEFAULT)

Manually

Remove the existing ZTPServer files:

```
rm -rf /usr/share/ztpserver/actions/*
rm -rf /usr/share/ztpserver/bootstrap/*
rm -rf /usr/lib/python2.7/site-packages/ztpserver*
rm -rf /bin/ztps*
rm -rf /home/ztpuser/ztpserver/ztpserver.egg-info/
rm -rf /home/ztpuser/ztpserver/build/*
```

Go to the ZTPServer directory where you previously cloned the GitHub repository, update it, then build and install the server:

```
bash-3.2$ git pull
bash-3.2$ python setup.py build
bash-3.2$ python setup.py install
```

My server keeps failing to load my resource files. What's going on?

Did you know?

```
a:b is INVALID YAML
a: b is VALID YAML syntax
```

Check out YAML syntax checker for more.

How do I validate the format of my config files?

To validate config files use `ztps --validate`:

```
[ztpsadmin@ztps ~]$ ztps --validate
Validating neighbordb ('/usr/share/ztpserver/neighbordb')... Ok!

Validating definitions...
Validating /usr/share/ztpserver/definitions/torb-withImageUpgrade... Ok!
Validating /usr/share/ztpserver/definitions/torb... Ok!
Validating /usr/share/ztpserver/definitions/tora-withImageUpgrade... Ok!
Validating /usr/share/ztpserver/definitions/tora... Ok!

Validating resources...
Validating /usr/share/ztpserver/resources/tor_hostnames... Ok!
Validating /usr/share/ztpserver/resources/ip_loopback... Ok!
Validating /usr/share/ztpserver/resources/ip_vlan100... Ok!
Validating /usr/share/ztpserver/resources/mgmt_subnet... Ok!

Validating nodes...
Validating /usr/share/ztpserver/nodes/001122334456/pattern... Ok!
Validating /usr/share/ztpserver/nodes/001122334456/definition... Ok!
Validating /usr/share/ztpserver/nodes/001122334455/pattern... Ok!
Validating /usr/share/ztpserver/nodes/001122334455/definition... Ok!
Validating /usr/share/ztpserver/nodes/001122334457/pattern... Ok!
Validating /usr/share/ztpserver/nodes/001122334457/definition... Ok!
```

How do I debug the ZTP Server provisioning process?

- If ZTP Server is running via wsgi, Check the Apache log files. Separate log files can be designated for ZTP Server's wsgi with the following:

```
<VirtualHost *:8080>
    CustomLog logs/ztpserver-access_log common
    ErrorLog logs/ztpserver-error_log
    ...
</VirtualHost>
```

- Run the standalone ZTP Server binary in debug mode and log the output to a file: `ztps --debug 2>&1 | tee ztps.log`
- After changing configuration directives in neighbordb, a definition, etc, you may need to remove the node directory of the node-under-test before retrying ZTP on the node. This will ensure that ZTP Server matches the node against neighbordb instead of `nodes/<serialnum>/pattern`.
- The `bootstrap` script may be manually run from a switch instead of going through an entire reload/ZTP cycle. To do this, download the script to the switch, then run it locally:

```
switch# bash wget http://ztpserver:8080/bootstrap
switch# bash chmod +x bootstrap
switch# bash sudo ./bootstrap
```

- On the client side, make sure you use XMPP (best) or remove syslog (second best) logging - you can configure that in *bootstrap.conf*.
- When requesting support, please include the output from the server (running in debug mode) and the console/log output from the switch.

How do I disable / enable ZTP mode on a switch

By default, any switch that does not have a `startup-config` will enter ZTP mode to attempt to retrieve one. This feature was introduced in EOS 3.7 for fixed devices and EOS 4.10 for modular ones. In ZTP mode, the switch sends out DHCP requests on all interfaces and **will not forward traffic** until it reboots with a config.

To cancel ZTP mode, login as admin and type `zerotouch cancel`. **This will trigger an immediate reload** of the switch, after which the switch will be ready to configure manually. At this point, if you ever erase the startup-config and reload, the switch will edn up ZTP mode again.

To completely disable ZTP mode, login as admin and type `zerotouch disable`. **This will trigger an immediate reload** of the switch after which the switch will will be ready to configure manually. If you wish to re-enable ZTP, go to configure mode and run `zerotouch enable`. The next time you erase the startup-config and reload the switch, the switch will end up ZTP mode again.

Note: vEOS instances come with a minimal startup-config so they do not boot in to ZTP mode by default. In order to use vEOS to test ZTP, enter `erase startup-config` and reload.

How can I test ZTPServer without having to reboot the switch every time?

From a bash shell on the switch:

```
# retrieve the bootstrap file from server
wget http://<ZTP_SERVER>:<PORT>/bootstrap
# make file executable
sudo chmod 777 bootstrap
# execute file
sudo ./bootstrap
```

What is the recommended test environment for ZTPServer?

The best way to learn about and test a ZTPServer environment is to build the server and virtual (vEOS) nodes with Packer. See https://github.com/arista-eosplus/packer-ztpserver for directions.

If you setup your own environment, the following recommendations should assist greatly in visualizing the workflow and troubleshooting any issues which may arise. The development team strongly encourages these steps as Best Practices for testing your environment, and, most of these recommendations are also Best Practices for a full deployment.

- During testing, only - run the standalone server in debug mode: `ztps --debug` in a buffered shell. NOTE: do NOT use this standalone server in production, however, except in the smallest environments (Approx 10 nodes or less, consecutively).
- Do not attempt any detailed debugging from a virtual or serial console. Due to the quantity of information and frequent lack of copy/paste access, this if often painful. Both suggested logging methods, below, can be configured in the *Bootstrap configuration*.
  - (BEST) Setup XMPP logging. There are many XMPP services available, including ejabberd, and even more clients, such as Adium. This will give you a single pane view of what is happening on all of your test switches. Our demo includes ejabberd with the following configuration:
    - Server: im.ztps-test.com (or your ZTPServer IP)
    - XMPP admin user: ztpsadmin@im.ztps-test.com, passwd eosplus
  - (Second) In place of XMPP, splecify a central syslog server in the bootstrap config.

How do I override the default system-mac in vEOS?

Add the desired MAC address to the first line of the file /mnt/flash/system_mac_address, then reboot (Feature added in 3.13.0F)

```
[admin@localhost ~]$ echo 1122.3344.5566 > /mnt/flash/system_mac_address
```

How do I override the default serial number or system-mac in vEOS?

As of vEOS 4.14.0, the serial number and system mac address can be configured with a file in /mnt/flash/veos-config. After modifying SERIALNUMBER or SYSTEMMACADDR, a reboot is required for the changes to take effect.

```
SERIALNUMBER=ABC12345678
SYSTEMMACADDR=1122.3344.5566
```

Internals

Implementation Details

Client-side implementation details

Action attributes

The bootstrap script will pass in as argument to the main method of each action a special object called 'attributes'. The only API the action needs to be aware for this object is the 'get' method, which will return the value of an attribute, as configured on the server:

- the value can be local to a particular action or global
- if an attribute is defined at both the local and global scopes, the local value takes priority
- if an attribute is not defined at either the local or global level, then the 'get' method will return **None**

e.g. (action code)

```
def main(attributes):
    print attributes.get('software_image')
```

Besides the values coming from the server, a couple of **special entries*** (always upper case) are also contained in the attributes object:

- 'NODE': a node object for making eAPI calls to localhost. See the *Bootstrap Client* documentation.

e.g. (action_code)

```
def main(attributes):
    print attributes.get('NODE').api_enable_cmds(['show version'])
```

Bootstrap URLs

1. DHCP response contains the **URL pointing to the bootstrap script** on the server
2. The location of the server is hardcoded in the bootstrap script, using the SERVER global variable. The bootstrap script uses this base address in order to generate the **URL to use in order to GET the logging details**: `BASE_URL/config` e.g.

   ```
   SERVER = 'http://my-bootstrap-server:80'    # Note that the port and the transport␣
   ↪mechanism
                                               # is included in the URL
   ```

3. The bootstrap script uses the SERVER base address in order to compute the **URL to use in order to POST the node's information:** `BASE_URL/config`

4. The bootstrap script uses the 'location' header in the POST reply as the **URL to use in order to request the definition**
5. **Actions and resources URLs**& are computed by using the base address in the bootstrap script: BASE_URL/actions/, BASE_URL/files/

Client - Server API

URL Endpoints

HTTP Method	URI
GET	/bootstrap
GET	/bootstrap/config
POST	/nodes
GET	/nodes/{id}
PUT	/nodes/{id}/startup-config
GET	/nodes/{id}/startup-config
GET	/actions/{name}
GET	/files/{filepath}
GET	/meta/{actions|files|nodes}/{PATH_INFO}

GET bootstrap script

GET /bootstrap

Returns the default bootstrap script

Request

```
GET /bootstrap HTTP/1.1
```

Response

```
Content-Type: text/x-python
<contents of bootstrap client script>
```

Response Headers

- Content-Type – text/x-python

Status Codes

- 200 OK – OK

Note: For every request, the bootstrap controller on the ZTPServer will attempt to perform the following string replacement in the bootstrap script): **"$SERVER" —> the value of the "server_url" variable in the server's global configuration file**. This string replacement will point the bootstrap client back to the server in order to enable the client to make additional requests for further resources on the server.

- if the `server_url` variable is missing from the server's global configuration file, 'http://ztpserver:8080' is used by default
- if the `$SERVER` string is missing from the bootstrap script, the controller will log a warning message and continue

GET bootstrap logging configuration

GET /bootstrap/config

Returns the logging configuration from the server.

Request

```
GET /bootstrap/config HTTP/1.1
```

Response

```
Content-Type: application/json
{
    "logging"*: [ {
        "destination": "file:/<PATH>" | "<HOSTNAME OR IP>:<PORT>",    //localhost␣
↪enabled
                                                                      //by default
        "level"*:        <DEBUG | CRITICAL | ...>,
    } ]
},
    "xmpp"*:{
        "server":           <IP or HOSTNAME>,
        "port":             <PORT>,                 // Optional, default 5222
        "username"*:        <USERNAME>,
        "domain"*:          <DOMAIN>,
        "password"*:        <PASSWORD>,
        "nickname":         <NICKNAME>,             // Optional, default␣
↪'username'
        "rooms"*:           [ <ROOM>, ... ]
        }
    }
}
```

Note: * Items are mandatory (even if value is empty list/dict)

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – OK

POST node details

Send node information to the server in order to check whether it can be provisioned.

POST /nodes

Request

```
Content-Type: application/json
{
    "model"*:              <MODEL_NAME>,
    "serialnumber"*:       <SERIAL_NUMBER>,
    "systemmac"*:          <SYSTEM_MAC>,
    "version"*:            <INTERNAL_VERSION>,
    "neighbors"*: {
        <INTERFACE_NAME(LOCAL)>: [ {
            'device':              <DEVICE_NAME>,
            'remote_interface':    <INTERFACE_NAME(REMOTE)>
        } ]
    },
}
```

Note: * Items are mandatory (even if value is empty list/dict)

Response

Status: 201 Created OR 409 Conflict will both return:

```
Content-Type: text/html
Location: <url>
```

Status Codes

- 201 Created – Created
- 409 Conflict – Conflict
- 400 Bad Request – Bad Request

GET node definition

Request definition from the server.

GET /nodes/ (*ID*)

Request

```
GET /nodes/{ID} HTTP/1.1
Accept: application/json
```

Response

```
Content-Type: application/json
{
    "name"*: <DEFINITION_NAME>
```

```
    "actions"*: [{ "action"*:          <NAME>*,
                "description":      <DESCRIPTION>,
                "onstart":          <MESSAGE>,
                "onsuccess":        <MESSAGE>,
                "onfailure":        <MESSAGE>,
                "always_execute":   [True, False],
                "attributes": { <KEY>: <VALUE>,
                                <KEY>: { <KEY> : <VALUE>},
                                <KEY>: [ <VALUE>, <VALUE> ]
                                }
                },...]
}
```

Note: * Items are mandatory (even if value is empty list/dict)

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – OK
- 400 Bad Request – Bad Request
- 404 Not Found – Not Found

PUT node startup-config

This is used to backup the startup-config from a node to the server.

PUT /nodes/ (*ID*) **/startup-config**

Request

```
Content-Type: text/plain
<startup-config contents>
```

Status Codes

- 201 Created – Created
- 400 Bad Request – Bad Request

GET node startup-config

This is used to retrieve the startup-config that was backed-up from a node to the server.

GET /nodes/ (*ID*) **/startup-config**

Request

```
Content-Type: text/plain
```

Response

Status: 201 Created OR 409 Conflict will both return:

```
Content-Type: text/plain
<startup-config contents>
```

Response Headers

- Content-Type – text/plain

Status Codes

- 200 OK – OK
- 400 Bad Request – Bad Request

GET actions/(NAME)

GET /actions/ (*NAME*)
Request action from the server.

Request Example

```
GET /actions/add_config HTTP/1.1
```

Response

```
Content-Type: text/x-python
<raw action content>
```

Response Headers

- Content-Type – text/x-python

Status Codes

- 200 OK – OK
- 404 Not Found – Not Found

GET resource files

GET /files/ (*RESOURCE_PATH*)
Request action from the server.

Request Examples

```
GET /files/images/vEOS.swi HTTP/1.1
GET /files/templates/ma1.template HTTP/1.1
```

Response

```
<raw resource contents>
```

:resheader Content-Type:text/plain :statuscode 200: OK :statuscode 404: Not Found

GET meta data for a resource or file

GET /meta/(actions|files|nodes)/ (*PATH_INFO*)
Request meta-data on a file.

Example Requests

```
GET /meta/actions/add_config HTTP/1.1
GET /meta/files/images/EOS-4.14.5F.swi HTTP/1.1
GET /meta/nodes/001122334455/.node HTTP/1.1
```

Response

```
{
  sha1: "d3852470a7328a4aad54ce030c543fdac0baa475"
  size: 160
}
```

:resheader Content-Type:application/json :statuscode 200: OK :statuscode 500: Server Error

Modules

Bootstrap Client

class **Node** (*server*)

Node object which can be used by actions via: attributes.get('NODE')

client
jsonrpclib.Server – jsonrpclib connect to Command API engine

api_config_cmds (*cmds*)
Run CLI commands via Command API, starting from config mode.

Commands are ran in order.

Parameters **cmds** (*list*) – List of CLI commands.

Returns

List of Command API results corresponding to the input commands.

Return type list

api_enable_cmds (*cmds*, *text_format=False*)
Run CLI commands via Command API, starting from enable mode.

Commands are ran in order.

Parameters

- **cmds** (*list*) – List of CLI commands.
- **text_format** (*bool, optional*) – If true, Command API request will run in text mode (instead of JSON).

Returns

List of Command API results corresponding to the input commands.

Return type list

append_rc_eos_lines (*lines*)
Add lines to rc.eos.

Parameters **lines** (*list*) – List of bash commands

append_startup_config_lines (*lines*)
Add lines to startup-config.

Parameters **lines** (*list*) – List of CLI commands

classmethod bash_cmds (*cmds*)
Executes bash commands in order - stops on first failure.

Parameters **cmds** – list of bash commands

Returns first failing command (None otherwise) code: exit code for first failing command (None otherwise) out: stdout for first failing command (None otherwise) err: stderr for first failing command (None otherwise)

Return type cmd

create_user (*user*, *group*, *passwd*, *root='/persist/local/'*, *ssh_keys=None*)
Create a local user on the bootstrapped node. If 'ssh_keys' are provided, they will be copied to $HOME/.ssh/authorized_keys. Also, rc.eos will be modified to add this user on every boot. If the user provided already exists, the function will continue and install the ssh_keys (if necessary). The $HOME/.ssh directory will be assigned 0700 permissions and the $HOME/.ssh/authorized_keys file will be assigned 0600 permissions in accord with SSH best practices.

Parameters

- **user** (–) – the username
- **group** (–) – the group assigned to the user
- **passwd** (–) – cleartext password
- **root** (–) – the path where the user's home directory will reside
- **ssh_keys** (–) – (optional) public keys that will be copied to ~$HOME/.ssh/authorized_keys

Raises

- ZtpError
- - *missing argument* – user, group, passwd
- **–** useradd fails, return the error
- **–** ssh_keys cannot be written to 'authorized_keys'
- **–** files cannot change ownership or permissions

Returns True if user created; False if otherwise

Return type bool

details ()
Get details.

Returns

System details

Format:

```
{'model':        <MODEL>,
 'version':      <EOS_VERSION>,
 'systemmac':    <SYSTEM_MAC>,
 'serialnumber': <SERIAL_NUMBER>,
 'neighbors':    <NEIGHBORS>        # see neighbors()
}
```

Return type dict

flash()
Get flash path.

Returns flash path

Return type string

has_startup_config()
Check whether startup-config is configured or not.

Returns True is startup-config is configured; false otherwise.

Return type bool

log_msg(*msg*, *error=False*)
Log message via configured syslog/XMPP.

Parameters

- **msg** (*string*) – Message
- **error** (*bool, optional*) – True if msg is an error; false otherwise.

neighbors()
Get neighbors.

Returns

LLDP neighbor

Format:

```
{'neighbors': {<LOCAL_PORT>:
 [{'device': <REMOTE_DEVICE>,
   'port': <REMOTE_PORT>}, ...],
...}}
```

Return type dict

rc_eos()
Get rc.eos path.

Returns rc.eos path

Return type string

retrieve_url(*url*, *path*)
Download resource from server.

If 'path' is somewhere on flash and 'url' points back to SERVER, then the client will request the metadata for the resource from the server (in order to check whether there is enogh disk space available). If 'url' points to a different server, then the 'content-length' header will be used for the disk space checks.

Raises ZtpError – resource cannot be retrieved: - metadata cannot be retrieved from server OR - metadata is inconsistent with request OR - disk space on flash is insufficient OR - file cannot be written to disk

Returns startup-config path

Return type string

classmethod server_address()
Get ZTP Server URL.

Returns ZTP Server URL.

Return type string

startup_config()
Get startup-config path.

Returns startup-config path

Return type string

classmethod substitute(*template*, *substitutions*, *strict=True*)
Perform variable substitution on a config template.

Parameters

- **template** (*string*) – EOS configuration template
- **substitutions** (*dict*) – set of substitutions for the template
- **strict** (*bool, optional*) – If true, method will raise Exception when template variables are missing from 'substitutions'.

Returns template string with variable substitution

Return type string

system()
Get system information.

Returns

System information

Format:

```
{'model':        <MODEL>,
 'version':      <EOS_VERSION>,
 'systemmac':    <SYSTEM_MAC>,
 'serialnumber': <SERIAL_NUMBER>}
```

Return type dict

Actions

- *add_config*
- *copy_file*
- *install_cli_plugin*
- *install_extension*
- *install_image*
- *replace_config*
- *send_email*
- *run_bash_script*
- *run_cli_commands*

add_config

main (*attributes*)

Appends config section to startup-config.

This action is dual-supervisor compatible.

url
path to source config/template

substitution_mode
loose|strict (default: loose)

variables
list of value substitutions

Special_attributes: NODE: API object - see documentation for details

Example

```
-
  action: add_config
  attributes:
    url: files/templates/ma1.template
    variables:
      ipaddress: allocate('mgmt_subnet')
  name: "configure ma1"
  onstart: "Starting to configure ma1"
  onsuccess: "SUCCESS: ma1 configure"
  onfailure: "FAIL: IM provisioning@example.com for help"
```

copy_file

main (*attributes*)

Copies file to the switch.

Copies file based on the values of 'src_url' and 'dst_url' attributes ('dst_url' should point to the destination folder).

This action is NOT dual-supervisor compatible.

src_url
path to source file

dst_url
path to destination

mode
octal mode for destination path

overwrite
replace|if-missing|backup (default: replace)

'overwrite' values:

- 'replace': the file is copied to the switch regardless of whether there is already a file with the same name at the destination;

- 'if-missing': the file is copied to the switch only if there is not already a file with the same name at the destination; if there is, then the action is a no-op;
- 'backup': the file is copied to the switch; if there is already another file at the destination, that file is renamed by appending the '.backup' suffix

Special_attributes: NODE: API object - see documentation for details

Example

```
-
  action: copy_file
  always_execute: true
  attributes:
    dst_url: /mnt/flash/
    mode: 777
    overwrite: if-missing
    src_url: files/automate/bgpautoinf.py
  name: "automate BGP peer interface config"
```

install_cli_plugin

main (*attributes*)

Installs CliPlugin.

This action is NOT dual-supervisor compatible.

url

path to the CliPlugin

Special_attributes: NODE: API object - see documentation for details

Example

```
-
  action: install_image
  always_execute: true
  attributes:
    url: files/my_cli_plugin
  name: "install cli plugin"
```

install_extension

main (*attributes*)

Installs extension.

If 'force' is set, then the dependency checks are overridden.

This action is NOT dual-supervisor compatible.

url

path to source extension file

force
ignore validation errors (default: false)

always_execute
perform copy even if file exists

Special_attributes: NODE: API object - see documentation for details

Example

```
-
  action: install_extension
  always_execute: true
  attributes:
    url: files/telemetry-1.0-1.rpm
  name: "Install Telemetry"
```

install_image

main (*attributes*)
Installs new software image.

If the current software image is the same as the 'version' attribute value, then this action is a no-op. Otherwise, the action will replace the existing software image.

For dual supervisor systems, the image on the active supervisor is used as reference.

This action is dual-supervisor compatible.

url
path to source image file

version
EOS version of new image file

downgrade
Boolean - Should EOS images be downgraded to match? (Default: True)

Special_attributes: NODE: API object - see documentation for details

Example

```
-
  action: install_image
  always_execute: true
  attributes:
    url: files/images/vEOS.swi
    version: 4.13.5F
    downgrade: true
  name: "validate image"
  onstart: "Starting to install image"
  onsuccess: "SUCCESS: 4.13.5F installed"
  onfailure: "FAIL: IM nick@example.com for help"
```

replace_config

main (*attributes*)

Replaces /mnt/flash/startup-config with new file.

This action is dual-supervisor compatible.

url
path to source config/template

Special_attributes: NODE: API object - see documentation for details

Example

```
-
  action: replace_config
  attributes:
    url: files/configs/tor-startup-config
  name: "tor config"
```

send_email

main (*attributes*)

Sends an email using an SMTP relay host

Generates an email from the bootstrap process and routes it through a smarthost. The parameters value expects a dictionary with the following values in order for this function to work properly.

```
{
    'smarthost':   <hostname of smarthost>,
    'sender':      <from email address>
    'receivers':   [ <array of recipients to send email to> ],
    'subject':     <subject line of the message>,
    'body':        <the message body>,
    'attachments': [ <array of files to attach> ],
    'commands':    [ <array of commands to run and attach> ]
}
```

The required fields for this function are smarthost, sender, and receivers. All other fields are optional.

This action is dual-supervisor compatible.

Parameters

- **attributes** (*list*) – list of attributes; use attributes.get(<ATTRIBUTE_NAME>) to read attribute values
- **node** (*internal*) – attributes.get('NODE') API: see documentation
- **smarthost** – hostname of smarthos>,
- **sender** – from email addres>
- **receivers** – [<array of recipients to send email to>]
- **subject** – subject line of the message
- **body** – the message body

- **attachments** – [<array of files to attach>]
- **commands** – [<array of commands to run and attach>]

Example

```
-
  action: send_mail
  attributes:
      smarthost: smtp.example.com
      from: noreply@example.com
      subject: This is a test message from a switch in ZTP
      receivers:
          bob@exmple.com
          helen@example.com
      body: Please see the attached 'show version'
      commands: show version
```

run_bash_script

main (*attributes*)

Runs a script in EOS from bash.

This action is dual-supervisor compatible.

url
path to source script/template

variables
optional – list of value substitutions (for a script template)

Special_attributes: NODE: API object - see documentation for details

Example

```
-
  action: run_bash_script
  attributes:
    url: files/scripts/install_script
    variables:
      version: 1.2.3
  name: 'install temp package'
```

run_cli_commands

main (*attributes*)

Runs a set of EOS commands, starting from enable mode.

This action is dual-supervisor compatible.

url
path to source command list/template

variables
optional – list of value substitutions (for a template)

Special_attributes: NODE: API object - see documentation for details

Example

```
-
  action: run_cli_commands
  attributes:
    url: files/templates/ma1.template
    variables:
      ipaddress: allocate('mgmt_subnet')
  name: 'configure ma1'
```

Glossary of terms

action an action is a Python script which is executed during the bootstrap process.

attribute an attribute is a variable that holds a value. attributes are used in order to customise the behaviour of actions which are executed during the bootstrap process.

definition a definition is a YAML file that contains a collection of all actions (and associated attributes) which need to run during the bootstrap process in order to fully provision a node

neighbordb neighbordb is a YAML file which contains a collection of patterns which can be used in order to map nodes to definitions

node a node is a EOS instance which is provisioned via ZTPServer. A node is uniquely identified by its unique_id (serial number or system MAC address) and/or unique position in the network.

pattern a pattern is a YAML file which describes a node in terms of its unique_id (serial number or system MAC) and/or location in the network (neighbors)

resource pool a resource pool is a is a set of resources which can be allocated on the server for the clients. For example, a YAML file can provide a mapping between a set or resources and the nodes to which some of the resources might have been allocated to (the nodes are uniquely identified via their system MAC).

unique_id the unique identifier for a node. This can be configured, globally, to be the serial number (default) or system MAC address in the ztpserver.conf file

Support

Contact

ZTPServer is an Arista-led open source community project. Users and developers are encouraged to contribute to the project. See CONTRIBUTING for more details.

Before requesting support, please collect the necessary data to include. See *Before Requesting Support*.

Commercial support may be purchased through your Arista account team.

Community-based support is available through:

- eosplus forum
- eosplus-dev@arista.com.
- IRC: irc.freenode.net#arista

Customization, and integration services are available through the EOS+ Consulting Services team at Arista Networks, Inc. Contact eosplus-dev@arista.com or your account team for details.

Known caveats

The authoritative state for any known issue can be found in GitHub issues.

- Only a single entry in a file-based resource pool may be allocated to a node (using the `allocate(resource_pool` plugin)).
- Users MUST be aware of the required EOS version for various hardware components (including transcievers). Neighbor (LLDP) validation may fail if a node boots with an EOS version that does not support the installed hardware. Moreoever, some EOS features configured via ZTPServer might be unsupported. Please refer to the Release Notes for more compatability information and to the Transceiver Guide .
- If a lot of nodes are being booted at the same time and they all share the same file-based resource files (using the `allocate(resource_pool` plugin)), retrieving the definition for each might be slow (5s or longer) if the resource files are very large. The workaround is to use another plugin or custom actions and allocate the resources from alternative sources (other than shared files) - e.g. SQL

Releases

The authoritative state for any known issue can be found in GitHub issues.

Release 1.5

New Modules

Enhancements

- **Booststrap client - use unix:sockets by default on images with support (344) [jerearista]**

- **Action install_image - support for disabling downgrade (343) [jerearista]**
- **Resolve file download issues - disable gzip, deflate in HTTP requests (342) [jerearista]**

Fixed

- **Fix "resource file missing" error on validate (339) [urvishpanchal]**
- **Docs - Fix setup.py install paths when on ReadTheDocs (328) [jerearista]**

Known Caveats

Release 1.4.1

(Published April, 2016)

The authoritative state for any known issue can be found in GitHub issues.

Bug fixes

- Pypi package missing plugins (332)

Release 1.4

(Published August, 2015)

The authoritative state for any known issue can be found in GitHub issues.

Enhancements

- Plugin infrastructure for resource pool allocation (121)
- Use the order of entries in the file for allocating resources from a file via the `allocate` plugin (319)
- Documenatation updates:
  - Plugin infrastructure for resource pool allocation (121)

Bug fixes

- Starting ZTPServer fails because `pkg_resources.DistributionNotFound: mock` (318)
- Bootstrap file cannot be read by server (308)
- Bootstrap script fails because of broken pipe in EOS-4.14.5+ (312)

Release 1.3.2

(Published March, 2015)

The authoritative state for any known issue can be found in GitHub issues.

Bug fixes

- **Prevented .node file from becoming corrupted on the server (298)**
- **Added .node filename to server-side logs (297)**
- **Change `refresh_ztps` script default to "master"** Refresh_ztps will, by default, update the installation to the latest released version. Previously, the default was to the development branch which may still be accomplished with `refresh_ztps --branch develop`.
- Fixes to RPM packaging:
  - **Quieted chcon during install (295)**
  - **Fixed issue where config files may not be kept during upgrade (296)**
  - **Fixed issue with native rpmbuild due to changes in handling VERSION (294)**
- Documentation updates:
  - **Troubleshooting chapter (272)**
  - **Additional content in the ZTP Server Cookbook (289)**
  - **ZTP Server benchmarking results**

Release 1.3.1

(Published February, 2015)

The authoritative state for any known issue can be found in GitHub issues.

Bug fixes

- fixes *pip* install/uninstall issues

Release 1.3

(Published February, 2015)

The authoritative state for any known issue can be found in GitHub issues.

Enhancements

- `ztps --validate` validates:
  - neighbordb syntax and patterns
  - resource files syntax
  - definition files syntax
  - pattern files syntax

```
$ ztps --validate
Validating neighbordb ('/usr/share/ztpserver/neighbordb')...
2015-01-13 18:03:55,006:ERROR:[validators:111] N/A: PatternValidator validation␣
↪error: missing attribute: definition
2015-01-13 18:03:55,006:ERROR:[validators:111] N/A: NeighbordbValidator validation␣
↪error: invalid patterns: set([(0, 's7151')])
```

```
ERROR: Failed to validate neighbordb patterns
Invalid Patterns (count: 1)
---------------------------
[0] s7151

Validating definitions...
Validating /usr/share/ztpserver/definitions/leaf.definition... Ok!
Validating /usr/share/ztpserver/definitions/leaf-no_vars.definition... Ok!

Validating resources...
Validating /usr/share/ztpserver/resources/leaf_man_ip... Ok!
Validating /usr/share/ztpserver/resources/leaf_spine_ip...
ERROR: Failed to validate /usr/share/ztpserver/resources/leaf_spine_ip
validator: unable to deserialize YAML data:
10.0.0.51/24: null
10.0.0.53/24: null
dfdsf dsfsd
10.0.0.54/24: JPE14140273

Error:
while scanning a simple key
in "<string>", line 3, column 1:
dfdsf dsfsd
could not found expected ':'
in "<string>", line 5, column 1:
10.0.0.54/24: JPE14140273
^

Validating nodes...
Validating /usr/share/ztpserver/nodes/JAS12170010/definition... Ok!
Validating /usr/share/ztpserver/nodes/JAS12170010/pattern... Ok!
```

- *run_bash_script* action allows users to run bash scripts during the bootstrap process
- *run_cli_commands* action allows users to run CLI commands during the bootstrap process
- *config-handlers* can be used in order to trigger scripts on the server on PUT startup-config request completion
- The auto **replace_config** action which is added to the definition whenever a startup-config file is present in a node's folder is now the first action in the definition which is sent to the client. This enables performing configuration updates during ZTR (Zero Touch Replacement) via 'always_execute' *add_config* actions in the definition file. One particularly interesting use-case is replacing one node with another one of a different model.
- `ztps --clear-resources` clears all resource allocations
- server-side logs are timestamped by default
- ZTP Server shows running version on-startup

```
# ztps
2015-02-09 16:50:35,922:INFO:[app:121] Starting ZTPServer v1.3.0...
...
```

Bug fixes

- upgrades/downgrades to/from v1.3+ will preserve the configuration files

  - *ztpserver.conf*, *ztpserver.wsgi*, *bootstrap.conf* and *neighbordb* are preserved (new default files are installed under <*filename*>.new)
  - all definitions, config-handlers, files, node folder, resources and files are preserved
  - *bootstrap* file, actions and libraries are always overwritten
- *bootstrap.conf* now supports specifying empty config sections:

```
logging:
  ...
xmpp:
```

```
logging:
xmpp:
  ...
```

Release 1.2

(Published December, 2014)

The authoritative state for any known issue can be found in GitHub issues.

Enhancements

- **Enhance neighbordb documentation (255)**
- **In case of failure, bootstrap cleanup removes temporary files that were copied onto switch during provisioning (253)**
- **"ERROR: unable to disable COPP" should be a warning on old EOS platforms (242)** A detailed warning will be displayed if disabling COPP fails (instead of an error).
- **Enhance documentation for open patterns(239)**
- **Document guidelines on how to test ZTPS (235)**
- **Document http://www.yamllint.com/ as a great resource for checking YAML files syntax (234)**
- **Make "name" an optional attribute in local pattern files (233)** node pattern file can contain only the interfaces directive now e.g.

```
interfaces:
- any:
    device: any
    port: any
```

- **Documentation should clarify that users must be aware of the EOS version in which certain transceivers were introduced**
- **Enhance the Apache documentation (231)**
- **Enhance documentation related to config files (229)**
- Disable meta information checks for remote URLs (224)
  - if URL points to ZTP server and destination is on flash, use metadata request to compute disk space (other metadata could be added here in the future)

  - it URL points to a remote server and destination is on flash, use 'content-length' to compute disk space - this will skip the metadata request

- Assume port 514 for remote syslog, if missing from bootstrap.conf (218)

  When configuring remote syslog destinations in bootstrap.conf, the port number is not mandatory anymore (if missing, a default value of 514 is assumed).

  e.g.

```
logging:
  - destination: pcknapweed
    level: DEBUG
```

- **Deal more gracefully with YAML errors in neighbordb (216)** YAML serialization errors are now exposed in ZTPS logs:

```
DEBUG: [controller:170] JPE14140273: running post_node
ERROR: [topology:83] JPE14140273: failed to load file: /usr/share/ztpserver/
↪neighbordb
ERROR: [topology:116] JPE14140273: failed to load neighbordb:
<b>expected a single document in the stream
  in "<string>", line 26, column 1:
    patterns:
    ^
but found another document
  in "<string>", line 35, column 1:
    ---
    ^</b>
DEBUG: [controller:182] JPE14140273: response to post_node: {'status': 400,
↪'body': '', 'content_type': 'text/html'}
s7056.lab.local - - [03/Nov/2014 21:05:33] "POST /nodes HTTP/1.1" 400 0
```

- **Deal more gracefully with DNS/connectivity errors while trying to access remote syslog servers (215)** Logging errors (e.g. bogus destination) will not be automatically logged by the bootstrap script. In order to debug logging issues, simply uncomment the following lines in the bootstrap script:

```
#---------------------------------SYSLOG----------------------
# Comment out this section in order to enable syslog debug
# logging
logging.raiseExceptions = False
#---------------------------------XMPP------------------------
```

  Example of output which is suppressed by default:

```
Traceback (most recent call last):
  File "/usr/lib/python2.7/logging/handlers.py", line 806, in emit
    self.socket.sendto(msg, self.address)
gaierror: [Errno -2] Name or service not known
Logged from file bootstrap, line 163
```

- **Make "name" an optional attribute in node definitions (214)** Definitions under /nodes/<NODE> do not have to have a 'name' attribute.

- **Increase HTTP timeout in bootstrap script (212)** HTTP timeout in bootstrap script is now 30s. https://github.com/arista-eosplus/ztpserver/issues/246 tracks making that configurable via bootstrap.conf. In the meantime, the workaround for changing it is manually editing the bootstrap file.

- **Remove fake prefixes from client and actions function names in docs (204)**

- **Tips and tricks - clarify vEOS version for both ways to set system MAC (203)**
- Enhance logging for "copy_file" action (187)
- **Local interface pattern specification should also allow management interfaces (185)** Local interface allows for:
  - management interface or interface range, using either mXX, maXX, MXX, MaXX, ManagementXX (where XX is the range)
  - management + ethernet specification on the same line: Management1-3,Ethernet3,5,6/7
- **Bootstrap script should cleanup on failure (176)**

```
$ python bootstrap --help
usage: bootstrap [options]

optional arguments:
  -h, --help            show this help message and exit
  --no-flash-factory-restore, -n
                        Do NOT restore flash config to factory defaul
```

  Added extra command-line option for the bootstrap script for testing.

  Default behaviour:

  - clear rc.eos, startup-config, boot-extensions (+folder) at the beginning of the process
  - in case of failure, delete all new files added to flash

  '-n' behaviour:

  - leave rc.eos, startup-config, boot-extensions (+folder) untouched
  - instead, bootstrap will create the new files corresponding to the above, with the ".ztp" suffix
  - never remove any files from flash at the end of the process, regardless of the outcome
- **Allow posting the startup-config to a node's folder, even if no startup-config is already present (169)**
- **Remove definition line from auto-generated pattern (102)** When writing the pattern file in the node's folder (after a neighbordb match):
  - 'definition' line is removed
  - 'variables' and 'node' are only written if non-empty
  - 'name' (that's the pattern's name) and 'interfaces' are always written

Fixed

- **server_url requires trailing slash "/" when adding subdirectory (244)**
- **Error when doing static node provisioning using replace_config (241)**
- **XMPP messages are missing the system ID (236)** XMPP messages now contain the serial number of the switch sending the message. 'N/A' is shown if the serial number is not available or empty.
- **Fix "node:" directive behaviour in neighbordb (230)**

  The following 'patterns' are now valid in neighbordb:

  - name, definition, node [,variables]
  - name, definition, interfaces [,variables]

  - name, definition, node, interfaces [,variables]

- **node.retrieve_resource should be a no-op if the file is already on the disk (225)** When computing the available disk space on flash for saving a file, the length of the file which is about to be overwritten is also considered.
- **Ignore content-type when retrieving a resource from a remote server or improve on the error message (222)** If a resource is retrieved from some other server (which is NOT the ZTPServer itself), then we allow any content-type.
- **ztpserver.wsgi is not installed by setup.py (220)**
- **ztps –validate broken in 1.1 (217)**

  ```
  ztps --validate PATH_TO_NEIGHBORDB
  ```

  can be used in order to validate the syntax of a neighbordb file.
- **install_extension action copies the file to the switch but doesn't install it (206)**
- **Bootstrap XMPP logging - client fails to create the specified MUC room (148)** In order for XMPP logging to work, a non-EOS user need to be connected to the room specified in bootstrap.conf, before the ZTP process starts. The room has to be created (by the non-EOS user), before the bootstrap client starts logging the ZTP process via XMPP.
- **ZTPS server fails to write .node because lack of permissions (126)**

Release 1.1

(Published August, 2014)

The authoritative state for any known issue can be found in GitHub issues.

Enhancements

- **V1.1.0 docs (181)** Documentation has been completely restructured and is now hosted at http://ztpserver.readthedocs.org/.
- **refresh_ztps - util script to refresh ZTP Server installation (177)** /utils/refresh_ztps can be used in order to automatically refresh the installation of ZTP Server to the latest code on GitHub. This can be useful in order to pull bug fixes or run the latest version of various development branches.
- **Et49 does not match Ethernet49 in neighbordb/pattern files (172)** The local interface in an interface pattern does not have to use the long interface name. For example, all of the following will be treated similarly: Et1, e1, et1, eth1, Eth1, ethernet1, Ethernet1.

  Note that this does not apply to the remote interface, where different rules apply.
- **Improve server-side log messages when there is no match for a node on the server (171)**
- **Improve error message on server side when definition is missing from the definitions folder (170)**
- **neighbordb should also support serialnumber as node ID (along with system MAC) (167)** Server now supports two types of unique identifiers, as specified in ztpserver.conf:

  ```
  # UID used in the /nodes structure (either systemmac or serialnumber)
  identifier = serialnumber
  ```

  The configuration is global and applies to a single run of the server (neighbordb, resource files, nodes' folders, etc.).

- **serialnumber should be the default identifier instead of systemmac (166)** The default identifier in ztpserver.conf is the serial number. e.g.

```
# UID used in the /nodes structure (either systemmac or serialnumber)
identifier = serialnumber
```

  This is different from v1.0, where the systemmac was the default.

- **Document which actions are dual-sup compatible and which are not (165)** All actions now document whether they are dual-sup compatible or not. See documentation for the details.
- **dual-sup support for install_image action (164)** install_image is now compatible with dual-sup systems.
- **Resource pool allocation should use the identifier instead of the systemmac (162)** The values in the resource files will be treated as either system MACs or serial numbers, depending on what identifier is configured in the global configuration file.
- **Document actions APIs (157)** The API which can be used by actions is now documented in the documentation for the bootstrap script module.
- **Get rid of return codes in bootstrap script (155)**
- **Bootstrap script should always log a detailed message before exiting (153)** bootstrap script will log the reason for exiting, instead of an error code.
- **Client should report what the error code means (150)**
- **Improve server logs when server does not know about the node (145)**
- **Configurable verbosity for logging options (server side) (140)** Bootstrap configuration file can now specify the verbosity of client-side logs:

```
...
xmpp:
username: ztps
password: ztps
domain: pcknapweed.lab.local
<b>msg_type : debug</b>
rooms:
    - ztps-room
```

  The allowed values are:

  - debug: verbose logging
  - info: log only messages coming from the server (configured in definitions)

  The information is transmitted to the client via the bootstrap configuration request:

```
####GET logging configuration
Returns the logging configuration from the server.

    GET /bootstrap/config

Request

    Content-Type: text/html

Response

    Status: 200 OK
    Content-Type: application/json
```

```
    {
        "logging"*: [ {
            "destination": "file:/<PATH>" | "<HOSTNAME OR IP>:<PORT>",    //
↪localhost enabled
                                                                          //by␣
↪default
            "level"*:        <DEBUG | CRITICAL | ...>,
        } ]
    },
        "xmpp"*:{
            "server":           <IP or HOSTNAME>,
            "port":             <PORT>,                 // Optional, default␣
↪5222
            "username"*:        <USERNAME>,
            "domain"*:          <DOMAIN>,
            "password"*:        <PASSWORD>,
            "nickname":         <NICKNAME>,             // REMOVED
            "rooms"*:           [ <ROOM>, ... ]
            "msg_type":         [ "info" | "debug" ]    // Optional, default␣
↪"debug"

        }
    }

>**Note**: * Items are mandatory (even if value is empty list/dict)
```

P.S. (slightly unrelated) The nickname configuration has been deprecated (serialnumber is used instead).

- **Configurable logging levels for xmpp (139)** bootstrap.conf:

```
logging:
...
xmpp:
...
nickname: ztps        // (unrelated) this was removed - using serial number␣
↪instead
msg_type: info        // allowed values ['info', 'debug']
```

If msg_type is set to 'info', only log via XMPP error messages and 'onstart', 'onsuccess' and 'onfailure' error messages (as configured in the definition).

- **Bootstrap should rename LLDP SysDescr to "provisioning" while executing or failing (138)**
- **Test XMPP for multiple nodes being provisioned at the same time (134)**
- **Server logs should include ID (MAC/serial number) of node being provisioned (133)** Most of the server logs will not be prefixed by the identifier of the node which is being provisioned - this should make debugging environments where multiple nodes are provisioned at the same time a lot easier.
- **Use serial number instead of system MAC as the unique system ID (131)**
- **Bootstrap script should disable copp (122)**
- **Bootstrap script should check disk space before downloading any resources (118)** Bootstrap script will request the meta information from server, whenever it attempts to save a file to flash. This information will be used in order to check whether enough disk space is available for downloading the resource.

```
####GET action metadata
Request action from the server.
```

```
    GET /meta/actions/NAME

Request

    Content-Type: text/html

Response

Status: 200 OK
    Content-Type: application/json
    {
        "size"*:  <SIZE IN BYTES>,
        "sha1": <HASH STRING>
    }

>**Note**: * Items are mandatory (even if value is empty list/dict)

    Status: 404 Not found
    Content-Type: text/html

    Status: 500 Internal server error                    // e.g.␣
↪permissions issues on server side
    Content-Type: text/html
```

- **ztps should check Python version and report a sane error is incompatible version is being used to run it (110)** ztps reports error if it is ran on a system with an incompatible Python version installed.
- **Do not hardcode Python path (109)**
- **Set XMPP nickname to serial number (106)** Serial number is used as XMPP presence/nickname. For vEOS instances which don't have one configured, systemmac is used instead.
- **Send serial number as XMPP presence (105)** Serial number is used as XMPP presence/nickname. For vEOS instances which don't have one configured, systemmac is used instead.
- **Support for EOS versions < 4.13.3 (104)** ZTP Server bootstrap script now supports any EOS v4.12.x or later.
- **neighbordb should not be cached (97)** Neighbordb is not cached on the server side. This means that any updates to it do not require a server restart anymore.
- **Definitions/actions should be loaded form disk on each GET request (87)** Definitions and actions are not cached on the server side. This means that any updates to them do not require a server restart anymore.
- **Fix all pylint warnings (83)**
- **add_config action should also accept server-root-relative path for the URL (71)** 'url' atrribute in add_config action can be either a URL or a local server path.
- **install_image action should also accept server-root-relative path for the URL (70)** 'url' atrribute in install_image action can be either a URL or a local server path.
- **Server logs should be timestamped (63)** All server-side logs now contain a timestamp. Use 'ztps –debug' for verbose debug output.
- **After installing ZTPServer, there should be a dummy neighbordb (with comments and examples) and a dummy resource**
- **need test coverage for InterfacePattern (42)**
- **test_topology must cover all cases (40)**

Resolved issues

- **Syslog messages are missing system-id (vEOS) (184)** All client-side log message are prefixed by the serial number for now (regardless of what the identifier is configured on the server).

  For vEOS, if the system does not have a serial number configured, the system MAC will be used instead.
- **No logs while executing actions (182)**
- **test_repository.py is leaking files (174)**
- **Allocate function will return some SysMac in quotes, others not (137)**
- **Actions which don't require any attributes are not supported (129)**
- **Static pattern validation fails in latest develop branch (128)**
- **Have a way to disable topology validation for a node with no LLDP neighbors (127)** COPP is disabled during the bootstrap process for EOS v4.13.x and later. COPP is not supported for older releases.
- **Investigate "No loggers could be found for logger sleekxmpp.xmlstream.xmlstream" error messages on client side (120)**
- **ZTPS should not fail if no variables are defined in neighbordb (114)**
- **ZTPS should not fail if neighbordb is missing (113)**
- **ZTPS installation should create dummy neighbordb (112)** ZTP Server install will create a placeholder neighbordb with instructions.
- **Deal more gracefully with invalid YAML syntax in resource files (75)**
- **Server reports AttributeError if definition is not valid YAML (74)**
- **fix issue with Pattern creation from neighbordb (44)**

Roadmap highlights

The authoritative state, including the intended release, for any known issue can be found in GitHub issues. The information provided here is current at the time of publishing but is subject to change. Please refer to the latest information in GitHub issues by filtering on the desired milestone.

Release 1.5

Target: January 2016

- topology-based ZTR (103)
- ZTPServer Cookbook - advanced topics (289)
- benchmark scale tests (261)

Release 2.0

Target: March 2016

- configure HTTP timeout in bootstrap.conf (246)
- all requests from the client should contain the unique identifier of the node (188)
- dual-sup support for install_extension action (180)

- dual-sup support for install_cli_plugin action (179)
- dual-sup support for copy_file action (178)
- action for arbitrating between MLAG peers (141)
- plugin infrastructure for resource pool allocation (121)
- md5sum checks for all downloaded resources (107)
- topology-based ZTR (103)

Tutorial

See https://eos.arista.com/quick-and-easy-veos-lab-setup/.

Other Resources

ZTPServer documentation and other reference materials are below:

- GitHub ZTPServer Repository
- ZTPServer wiki
- Packer VM build process
- ZTPServer Python (PyPI) package
- YAML Code Validator
- ZTPServer WSGI Benchmarking

Troubleshooting

Basics

When the ZTP process isn't behaving as expected, there are some basics that should be checked regularly.

Updating to the latest Release is strongly encouraged

ZTP Server is continually being enhanced and improved and its entirely possible that the issue you've encountered has already been addressed, either in the documentation such as *Tips and tricks*, or in the code, itself. Therefore, we strongly encourage anyone experiencing difficulty to reproduce the issue on the latest release version before opening an issue or requesting support. See *Upgrading*.

If the switch is not attempting Zero Touch Provisioning

Check whether ZTP has been disabled on the switch:

```
Arista#show zerotouch
```

Validate the ZTP Server configuration syntax

Many errors are simply due to typos or other syntax issues in config files. It is good practice to use the –validate option to ztps and to paste configs in to *http://yamllint.com/* to ensure they are well-formed YAML:

```
[user@ztpserver]$ ztps --validate-config
```

Other troubleshooting steps

A number of other troubleshooting steps including how to specify the separate apache log files just for ZTP Server, and how to do a test run of ztpserver without reloading a switch are located on the *Tips and tricks* page.

Before Requesting Support

Before requesting support, it is important to perform the following steps to collect sufficient data to reduce information requests and enable timely resolution.

Version and Install method

If not already recorded in the logs, please execute `ztps --version` and specify whether your installation was from source (github), pip, RPM, or a packer-ztpserver canned VM.

Server-side logs

The location of server-side logs may vary depending on your specific environment.

- If running ZTP Server via Apache, check the VirtualHost definition for CustomLog and ErrorLog entries, otherwise, look in the default Apache logs. On Fedora, those will be in /var/log/httpd/
- If running the standalone `ztps` binary, a good choice for debugging, please include the `--debug` option. Using `ztps --debug 2>&1 | tee ztpserver.log` will log the output to both the screen and a file.

Client-side logs

Ensure the bootstrap client is configured to log to syslog or XMPP via /usr/share/ztpserver/bootstrap/bootstrap.conf and include that output. Attempting to collect client side logs from the console frequently results in missing information due to scroll buffers or line length.

Configuration Files

Please, also, include the files in /etc/ztpserver/ and /usr/share/ztpserver/ directories. `tar czvf my_ztpserver_config.tgz /etc/ztpserver/ /usr/share/ztpserver/`

License

Third party

Requests v2.3.0: HTTP for Humans

HTTP Routing Table

/actions

/bootstrap

/files

/meta

/nodes

Python Module Index

a

c

Index

U

V