ZTPServer Documentation

Release 1.2.0

Arista Networks

December 05, 2014

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

2.1 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

2.1.1 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 stuartup-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

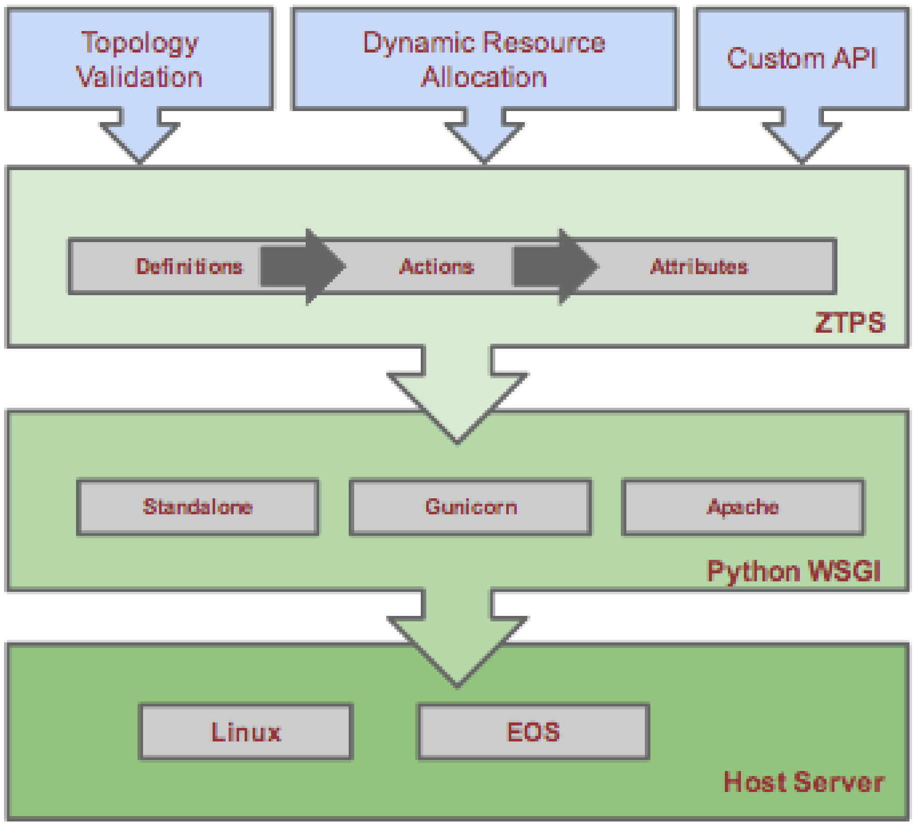
See the ZTP Tech Bulletin and the Press Release for more information on ZTP.

2.1.2 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)

2.1.3 Server



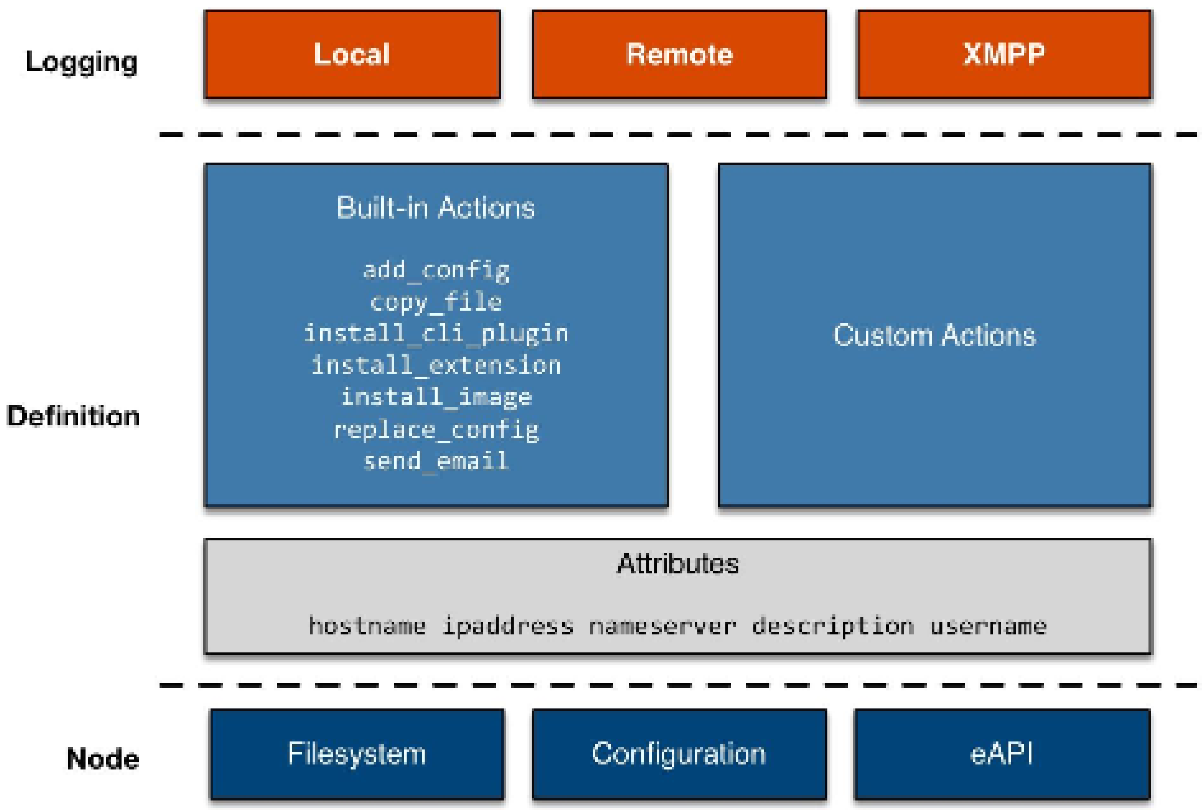
The server can run on any standard x86 server. Currently the only OS-es tested are Linux and MacOS, but theoretically any system that supports Python could run ZTPServer. The server provides a Python WSGI compliant interface, along with a standalone HTTP server. The built-in HTTP server runs by default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

2.1.4 Client



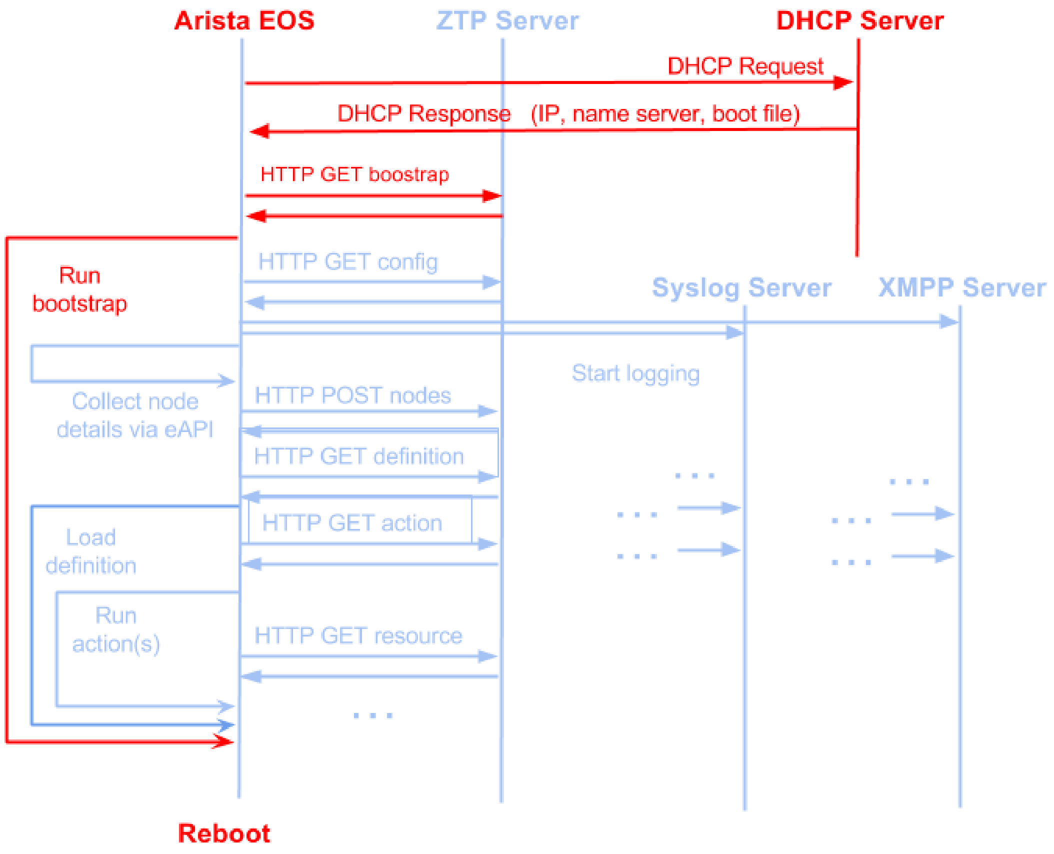
The client or **bootstrap file** is retrieved by the node via an HTTP GET request made to the ZTPServer (the URL of the file is retrieved via DHCP option 67). This file executes locally and gathers system and LLDP information from the node and sends it back to the ZTPServer. Once the ZTPServer processes 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.

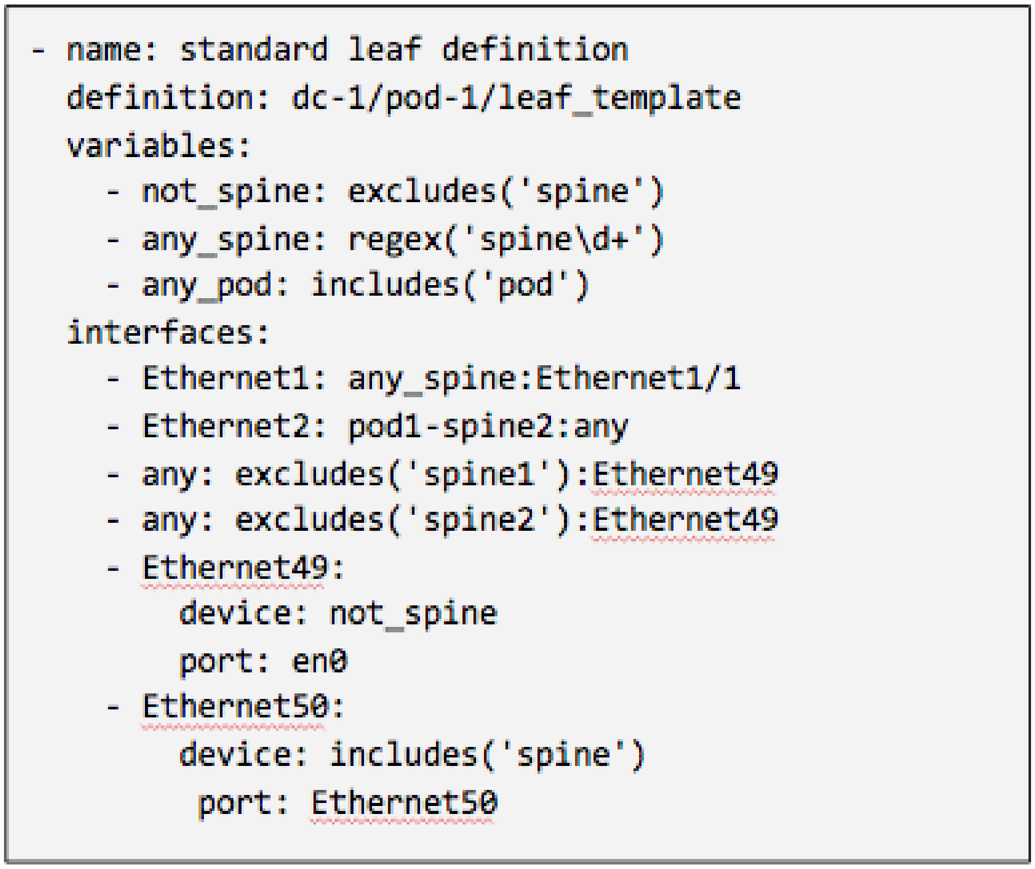
2.1.5 ZTP Client-Server Message Flows

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.)


Arista EOS
ZTP Server
DHCP Server
DHCP Request
DHCP Response (IP, name server, boot file)
HTTP GET boostrap
Run bootstrap
HTTP GET config
Syslog Server
XMPP Server
Start logging
Collect node details via eAPI
HTTP POST nodes
HTTP GET definition
HTTP GET action
Load definition
Run action(s)
HTTP GET resource
Reboot

2.1.6 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)

2.1.7 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

2.2 Installation

2.2.1 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.

2.2.2 Installation Options

- *Turn-key VM Creation*
- *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
...
```

2.2.3 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 status 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>";
  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.

2.3 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!

2.3.1 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.

2.3.2 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 FILENAME    Runs a validation check on neighbordb
--debug                Enables debug output to the STDOUT
```

When ZTPServer starts, it reads the path information to neighbordb and other files from the global configuration file. 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.

2.4 Configuration

2.4.1 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
            .node
            attributes
    actions/
    files/
    definitions/
    resources/
    neighbordb
```

2.4.2 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 FILENAME   Runs a validation check on neighbordb
  --debug               Enables debug output to the STDOUT
(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=/var/lib/ztpserver
data_root=<PATH>

# UID used in the /nodes structure (serialnum is not supported yet)
# 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>

# 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>

[files]
# Path for the files directory (overriding data_root/files)
# default=files
folder=<path>
# default=data_root (from above)
path_prefix=<path>

[actions]
# Path for the actions directory (overriding data_root/actions)
# default=actions
folder=<path>
# default=data_root (from above)
path_prefix=<path>

[bootstrap]
# Path for the bootstrap directory (overriding data_root/bootstrap)
# default=bootstrap
folder=<path>
# default=data_root (from above)
path_prefix=<path>

# Bootstrap filename
# 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.

2.4.3 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.

2.4.4 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 unique_id as the name
- create/symlink a startup-config or definition in the newly-created folder
- if topology validation is enabled, also create/symlink a pattern file

2.4.5 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:

- 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`
- 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 **last** in the definition.

2.4.6 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>
```

2.4.7 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:
    - 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 functions.

The supported functions are:

- **allocate(resource_pool)** - allocatea an available resource from a resource pool; the allocation is perform on the server side and the result of the allocation is passed to the client via the definition

Note: Functions can only be used 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.

2.4.8 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

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

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

2.4.9 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"
}
```

2.4.10 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/.

2.4.11 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 node 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.

```
variables:
    variable_name: function
...
patterns:
    - name: <single line description of pattern>
      definition: <defintion_url>
      node: <unique_id>
      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**

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

unique_id

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

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`: 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

2.4.12 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.

2.4.13 Actions

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

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

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.

2.4.14 Resource pools

`[data_root]/resources/` contains global resource pools from which attributes in definitions can be allocated via the allocate(...) function.

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 (whenever the allocate(...) function is used in the definition).

```
<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.

2.4.15 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.

2.5 Examples

2.5.1 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

# 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
```

```
[ files]
# Path for the files directory (overriding data_root/files)
folder = files
path_prefix = /usr/share/ztpserver

[actions]
# Path for the actions directory (overriding data_root/actions)
folder = actions
path_prefix = /usr/share/ztpserver

[bootstrap]
# Path for the bootstrap directory (overriding data_root/bootstrap)
folder = bootstrap
path_prefix = /usr/share/ztpserver

# Bootstrap filename
filename = bootstrap

[neighbordb]

# Neighbordb filename (file located in data_root)
filename = neighbordb
```

2.5.2 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
```

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

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

2.5.4 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
```

2.5.5 Sample templates

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

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

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

2.5.6 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
```

2.5.7 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

2.5.8 More examples

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

2.6 Tips and tricks

2.6.1 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
```

2.6.2 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.

2.6.3 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.

2.6.4 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
```

2.6.5 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.

2.6.6 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
```

2.6.7 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
```

2.7 Internals

2.7.1 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/

2.7.2 Client - Server API

URL Endpoints

HTTP Method	URI
GET	/bootstrap/config
GET	/bootstrap
POST	/nodes
PUT	/nodes/{id}
GET	/nodes/{id}
GET	/actions/{name}
GET	/files/{filepath}

GET bootstrap script

GET /bootstrap

Returns the default bootstrap script

Response

```
Status: 200 OK
Content-Type: text/x-python
```

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 logging configuration

GET /bootstrap/config

Returns the logging configuration from the server.

Request

```
GET /bootstrap/config HTTP/1.1
Host:
Accept:
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>,             // Optional, default 'username'
        "rooms"*:           [ <ROOM>, ... ]
        }
    }
}
```

Note: * Items are mandatory (even if value is empty list/dict)

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
Content-Type: text/html
Location: <url>

Status: 409 Conflict
Content-Type: text/html
Location: <url>

Status: 400 Bad Request
Content-Type: text/html
```

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
Host:
Accept: applicatino/json
Content-Type: text/html
```

Response

```
Status: 200 OK
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)

Status Codes

- 400 Bad Request – Bad Request
- 404 Not Found – Not Found

GET action

GET /actions/(*NAME*)

Request action from the server.

Request

```
Content-Type: text/html
```

Response

```
Content-Type: text/x-python
```

Status Codes

- 200 OK – OK
- 400 Bad Request – Bad Request
- 404 Not Found – Not Found

Status: 200 OK Content-Type: text/plain <PYTHON SCRIPT>

Status: 200 Bad request Content-Type: text/x-python

GET resource

GET /files/(*RESOURCE_PATH*)

Request action from the server.

Request

```
Content-Type: text/html
```

Response

```
Status: 200 OK
Content-Type: text/plain
<resource>
```

Status Codes

- 200 OK – OK
- 404 Not Found – Not Found

2.7.3 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

`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

`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`

`add_config`

`main``(`*attributes*`)`

Adds startup-config section.

Appends config section to startup config based on the value of the 'url' attribute.

This action is dual-supervisor compatible.

Accepts: A list of attributes; use attributes.get(<ATTRIBUTE_NAME>) to read attribute values

Parameters

- **url** – path to config snippet/template
- **substitution_mode** – loose|strict Default: loose
- **variables** – A list of variable: value substitutions
- **Special_attributes** – node: attributes.get('NODE') API: see documentation

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 detination folder). If 'overwrite' is set to:

- '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

If 'overwrite' is not set, then 'replace' is the default behaviour.

This action is NOT dual-supervisor compatible.

Parameters

- **attributes** – list of attributes; use attributes.get(<ATTRIBUTE_NAME>) to read attribute values
- **node** (*internernal*) – attributes.get('NODE') API: see documentation
- **src_url** – Source location
- **dst_url** – Destination
- **mode** – Octal mode
- **overwrite** – Overwrite existing files? <replace|if-missing|backup> Default: replace

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 EOS CliPlugin.

Installs CliPlugin based on the value of the 'url' attribute.

This action is NOT 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
- **url** – path to the cli plugin

Example

```
-
  action: install_image
  always_execute: true
  attributes:
    url: files/my_cli_plugin
  name: "install cli plugin"
```

`install_extension`

main(*attributes*)

Installs EOS extension.

Installs extension based on the value of the 'url' attribute. If 'force' is set, then the dependency checks are overridden.

This action is NOT 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
- **url** – path to the rpm or swix extension

- **force** – Force installation regardless of checks. Default: False

Example

```
-
  action: install_image
  always_execute: true
  attributes:
    url: files/telemetry-1.0-1.rpm
  name: "Install Telemetry"
```

`install_image`

main (*attributes*)

Installs software image on the switch.

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.

Parameters

- **attributes** (*list*) – list of attributes; use attributes.get(<ATTRIBUTE_NAME>) to read attribute values
- **node** (*internal*) – attributes.get('NODE') API: see documentation
- **url** – path to .swi file
- **version** – target version of the .swi file.

Example

```
-
  action: install_image
  always_execute: true
  attributes:
    url: files/images/vEOS.swi
    version: 4.13.5F
  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 stratup-config on the switch.

Replaces/adds /mnt/flash/startup-config based on the value of the 'url' attribute.

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
- **url** – path to config/template

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
```

2.8 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 YAML file which provides 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

2.9 Support

2.9.1 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.

Community-based support is available through:

- eosplus forum
- eosplus-dev@arista.com.
- IRC: irc.freenode.net#arista

Commercial support, customization, and integration services are available through the EOS+ team at Arista Networks, Inc. Contact eosplus-dev@arista.com for details.

2.9.2 Known caveats

The authoritative state for any known issue can be found in GitHub issues.

- v1.1: The management interfaces may not be used as valid local interface names in neighbordb. When creating patterns, use `any` instead. (fixed in v1.2)
- Only a single entry in a resource pool may be allocated to a node.
- 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 .

2.9.3 Releases

The authoritative state for any known issue can be found in GitHub issues.

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': '', 'cor
  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 ser
    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)**

2.9.4 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 1.3

Target: January 2015

- validate all YAML files via 'ztpserver –validate' (247)
- action which enables running arbitrary set of EOS CLI commands (211)
- show server's version in startup logs (207)
- command-line option for clearing resource pools (163)
- hook to run script after posting files on the server (132)
- action which enables running arbitrary bash commands (108)

Release 2.0

Target: March 2015

- 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)

2.9.5 Video tutorial

See http://www.youtube.com/playlist?list=PL6kEnPnH7OA4oc5jzhUW0ivVX1sMdfNpV.

2.9.6 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

2.10 License

2.10.1 Third party

Requests v2.3.0: HTTP for Humans

Python Module Index

Index