#ANSIBLEFEST2019

Deep dive with Network connection plugins

Ganesh B. Nalawade
PSE, Ansible Engineering
Github/IRC: ganeshrn

Nathaniel Case
SSE, Ansible Engineering
Github/IRC: Qalthos

Red Hat | ANSIBLE

Share your automation story - Nathaniel

1. How did you get started with Ansible?

First for hobby projects, but I really started to dive in when I inherited a monster automation project in another tool that no one could fix right. Less than a day later, it had been redone from scratch in Ansible and not only worked but had new capabilities we'd been needing.

2. How long have you been using it?

Around five years now, probably starting with 1.8.
3. What's your favorite thing to do when you Ansible?

Just being able to get moving quickly is my favorite part.

Share your automation story - Ganesh

1. How did you get started with Ansible?

Previously worked on network management plane for networking vendors, designing and developing on/off box network automation solution and that's how I was introduced with Ansible. I started as Ansible community contributor in 2016 and joined Ansible Engineering in April 2017 to work full time mainly on Ansible networking.

2. How long have you been using it?

More than three years now

3. What's your favorite thing to do when you Ansible?

Adding new features in Ansible code to enhance network automation user experience with Ansible

AGENDA

- Why persistent connection?
- How persistent connection works?
- How network connection plugins work with persistent framework?
- Deep dive with network connection plugins

Why persistent connection?

Top level view of Ansible how Ansible works:

1. Parse and load inventory
2. Parse and load playbooks
3. for each play in the playbooks:
    for each task in the play:
        for each host (filtered on play):
            run the task on the host and read the results

The task by default runs in the forked worker process

Why persistent connection?...(contd.)

Ansible modules execution - Example of Linux vs Network host

Linux host:

1) By default (when pipelining is disabled) Ansible will open ssh connection to the remote host
   a) create temporary directory
   b) Create a tar of the local module file, task arguments, boilerplate code etc and deliver to the target host (when using python) into a the temporary directory
   c) execute the module, read the return result in JSON format
      and delete the temporary directory on the remote host

2) Since module is executed remotely it requires an executable environment on remote host.

3) Ansible uses ssh “ControlPersist” feature wherever applicable to persist ssh session across tasks.

Why persistent connection?...(contd.)

Ansible modules execution - Example of Linux vs Network host

Network host:

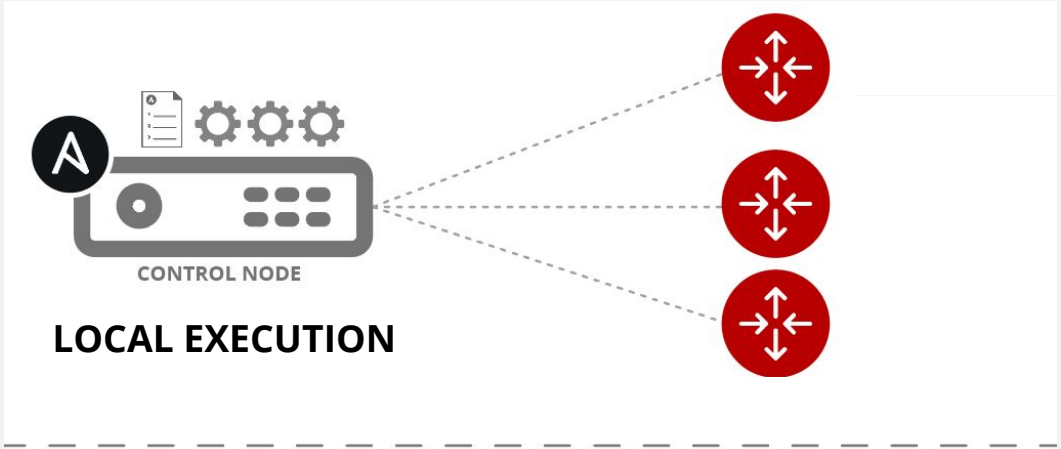
1) Most network device does not have a remote execution environment installed locally on the network os.

2) That's why Ansible network modules are not copied to target host for remote execution instead they run locally on the control node.

3) For network_cli/netconf connection types Ansible establishes an SSH transport to the target device and then open an SSH channel (subsytem) to send and receive data.

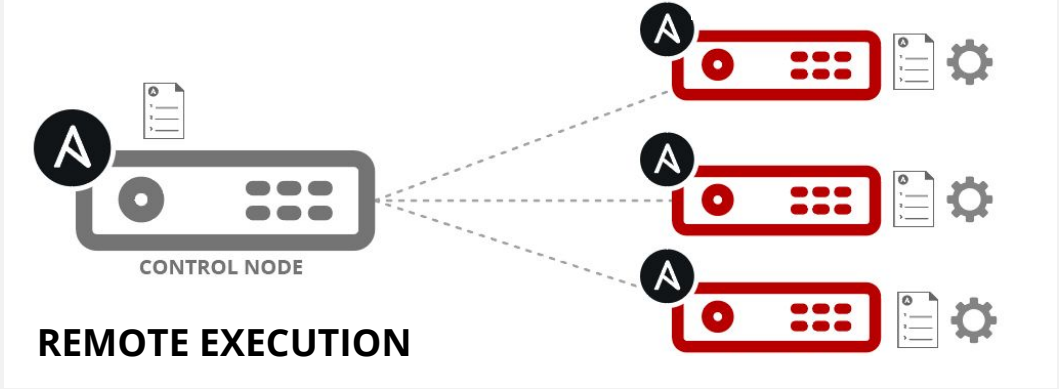
4) For httpapi connection type Ansible uses urllib2 under the hood to communicate with the remote host.

Module code is executed locally on the control node

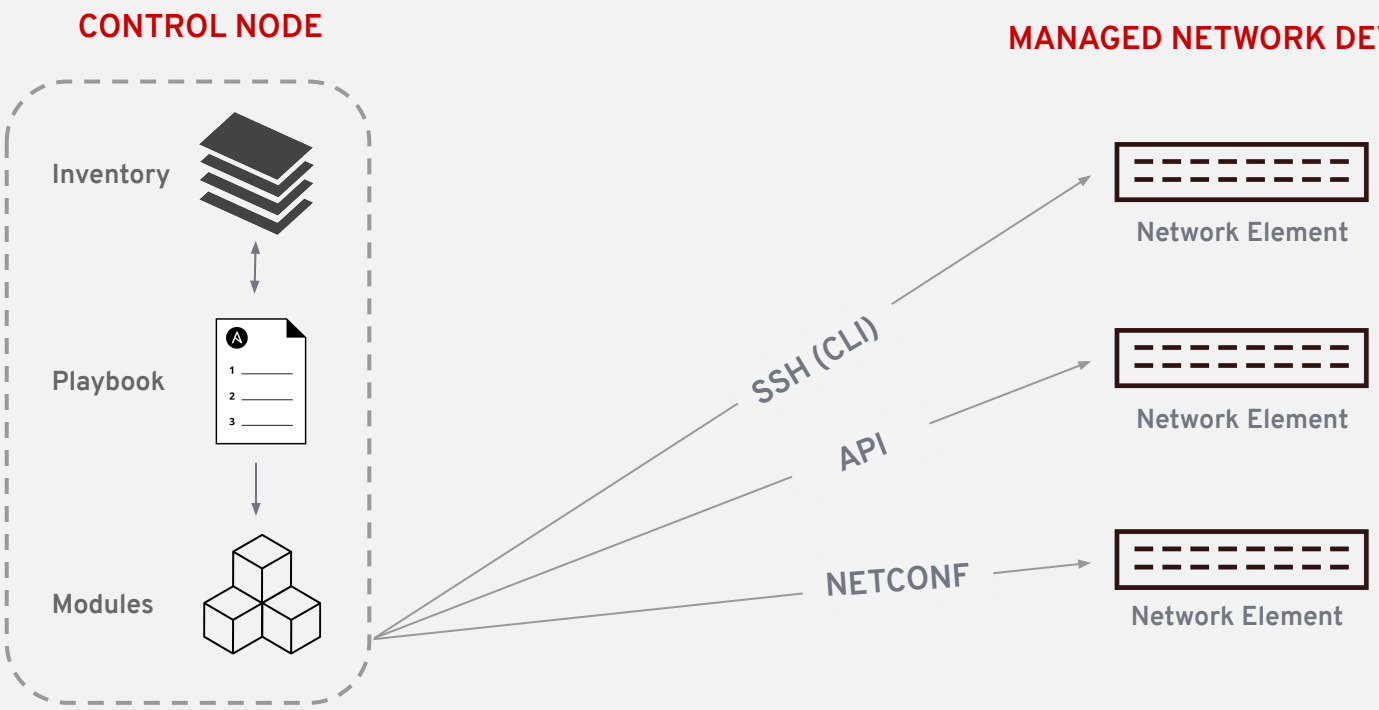


NETWORKING DEVICES

Module code is copied to the managed node, executed, then removed



LINUX/WINDOWS HOSTS



Managed Nodes (Inventory):
A collection of endpoints being managed via SSH or API.

Control Node:
Any client system (server, laptop, VM) running Linux or Mac OSX

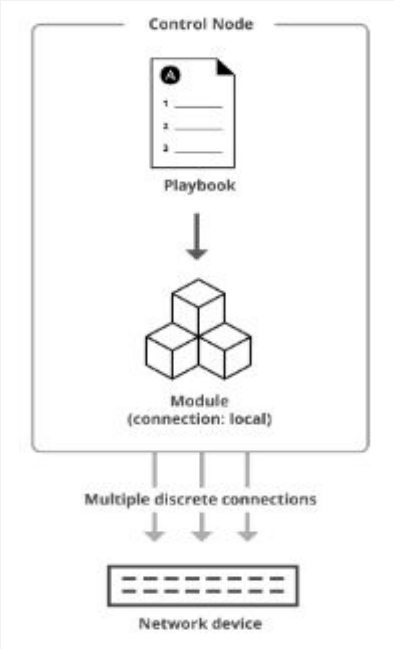
Modules:
Handles execution of remote system commands

Why persistent connection?...(contd.)

1) Without persistent connection for each task which run in it's own worker process Ansible will have to create a new connection with remote host, execute the module, read the result and close the connection.

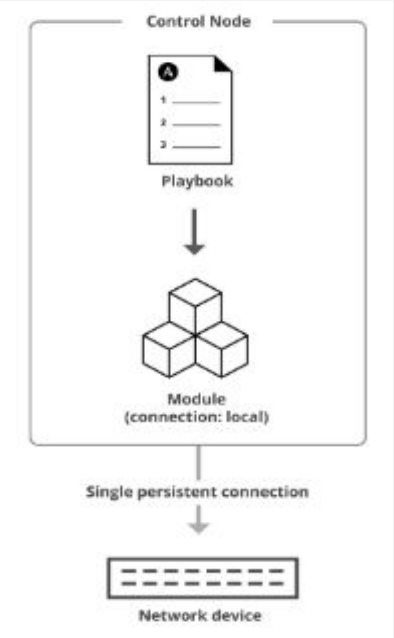
2) For small number of task this may not be a concern but as the number of task increases the time required to create and tear down the connection increases drastically.

Ansible 2.2 and earlier



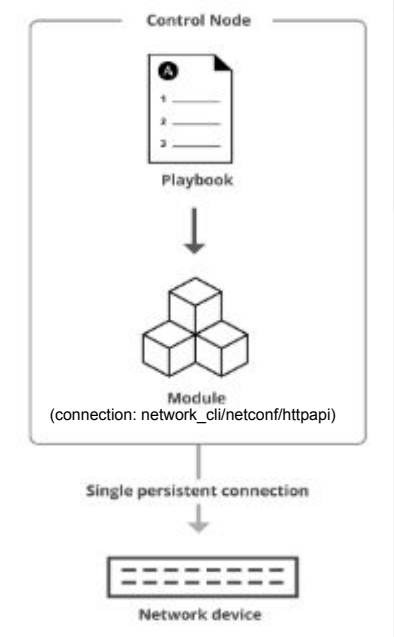

```
provider:
  host: xxxx
  username: xxxx
  password: xxxx
  port: xxxx
  authorize: True
```

Ansible 2.3, 2.4




```
provider:
  host: xxxx
  username: xxxx
  password: xxxx
  port: xxxx
  authorize: True
```

Ansible 2.5 and later




```
ansible_host: xxxx
ansible_user: xxxx
ansible_password: xxxx
ansible_port: xxxx
ansible_become: True
```

How persistent connection works?

1) Task executor (executor/task_executor.py) loads the connection object based on the value of ***"ansible_connection"*** variable.

2) If value of ***"force_persistence"*** is set to True in the connection class the task_executor will start ***"ansible-connection"*** which runs in background as a daemon process.

3) ansible-connection creates a local domain socket which is a hash of remote address, port, remote_user, connection type and ansible-playbook parent process id.

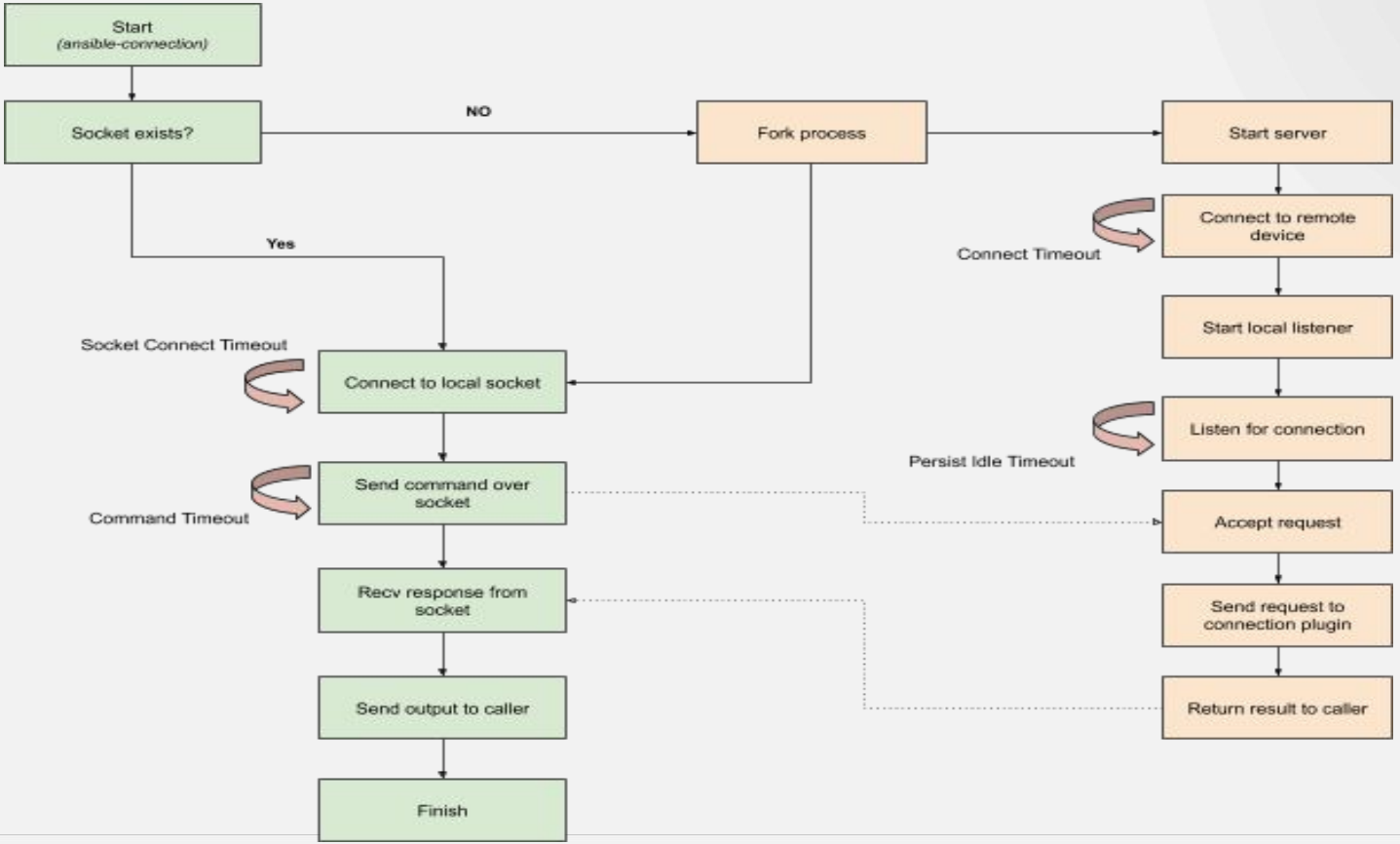
4) This local domain socket is used to communicate between module and connection (eg: plugins/connection/network_cli.py) side code.

How persistent connection works?...(contd.)

5) ansible-connection process also initiates a connection with remote host by invoking "***_connect()"*** method in the ***"Connection"*** class.

6) Since ansible-connection runs as a background process the connection with remote host is active for the duration of ansible-playbook run and persist across task runs till until either command_timeout or connect_timeout is triggered.

How persistent connection works?...(contd.)



How persistent connection works?...(contd.)

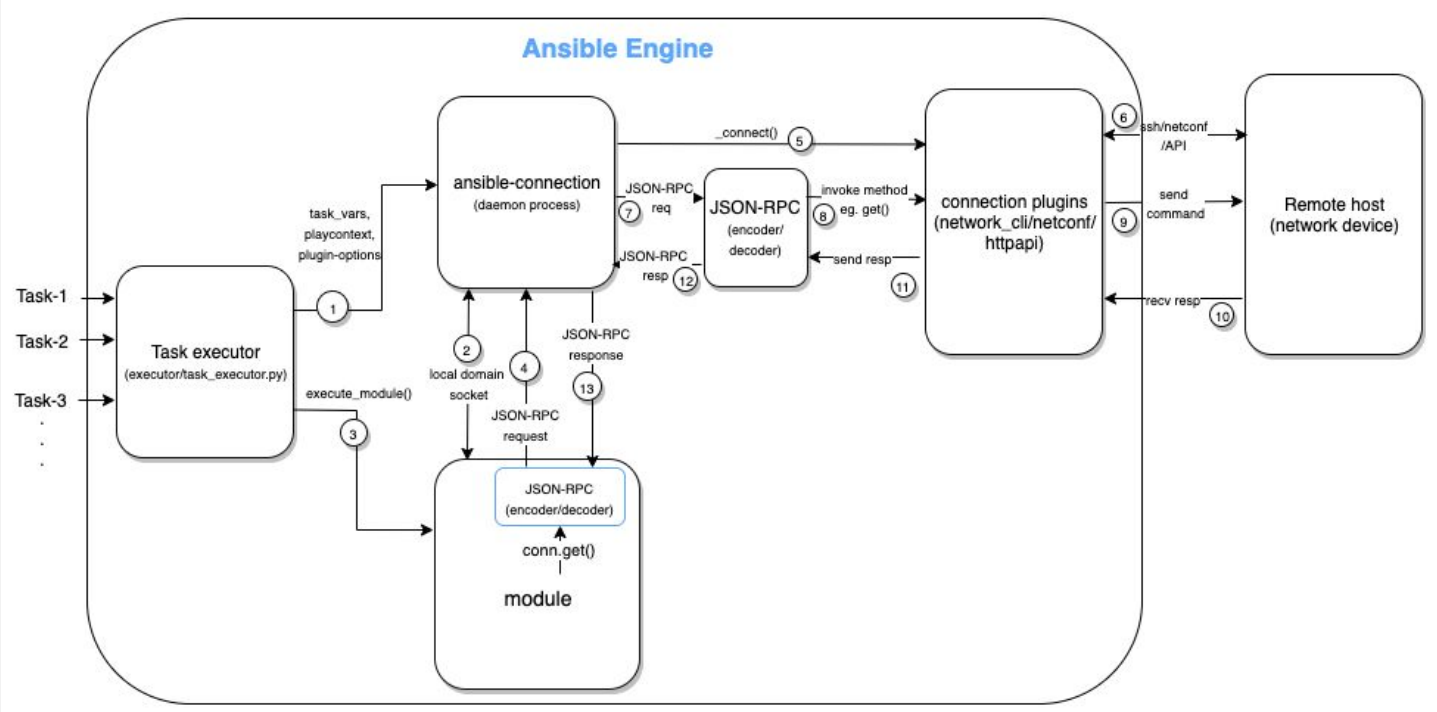
1) To execute commands on remote host the module invokes a method on module side Connection class object.

```
from ansible.module_utils.connection import Connection
<--snip-->
conn = Connection(module._socket_path)
conn.get(command='show version')
```

2) The Connection class invokes __rpc__() method (similar to method missing functionality) which encodes the method call into JSON-RPC 2.0 compliant JSON object (RPC request).

3) It consist of method name (eg: get), the parameters (eg: command='show version') and a unique id.

How persistent connection works?...(contd.)



Persistent framework Architecture

How persistent connection works?...(contd.)

JSON-RPC request (module → ansible-connection)

```
{
    "jsonrpc": "2.0",
    "method": "get",
    "id": "ebf2fb4e-a129-4aea-9f00-c0fca0100288",
    "params": [
        [], {
            "command": "show version",
            "newline": true,
            "sendonly": false,
            "check_all": false,
            "prompt": null,
            "answer": null
        }
    ]
}
```

JSON-RPC response (ansible-connection → module)

```
{
    "jsonrpc": "2.0",
    "id": "ebf2fb4e-a129-4aea-9f00-c0fca0100288",
    "result": "Cisco IOS Software, <--snip-->"
}
```

Deep dive with network connections

Connections and platform-specific plugins

1) Persistent connection plugins define how to use the transport, but platform-specific actions sometimes have to be taken

2) The connection will load another plugin based on the value of **“ansible_network_os”**
   a) For httpapi plugins/httpapi/nxos.py
   b) For network_cli
      i) plugins/terminal/ios.py
      ii) plugins/cliconf/ios.py
   c) For netconf plugins/netconf/iosxr.py

3) Methods defined in these plugins are available from the Connection class just as connection plugin methods are.

Connection type - httpapi

1) httpapi does not act like any of the other connections here.

2) httpapi is just like module_utils/urls.py… HTTP requests through httpapi even use urls.py

3) httpapi is a first-class connection- no provider needed.

4) httpapi is not specific to networking.

5) A well-constructed httpapi plugin should take in structured data and return structured data.

Connection type - httpapi...(contd.)

1) Introduced with Ansible 2.6
2) Initially only designed for EAPI & NX-API
3) Plugin and feature list grew over releases
4) 2.9 will have 12 httpapi plugins

Connection type - httpapi...(contd.)

```
CONTENT_TYPE = 'application/yang-data+json'


class HttpApi(HttpApiBase):
    def send_request(self, data, **message_kwargs):
        if data:
            data = json.dumps(data)

        path = '/'.join([self.get_option('root_path').rstrip('/'), message_kwargs.get('path', '').lstrip('/')])

        headers = {
            'Content-Type': message_kwargs.get('content_type') or CONTENT_TYPE,
            'Accept': message_kwargs.get('accept') or CONTENT_TYPE,
        }
        response, response_data = self.connection.send(path, data, headers=headers, method=message_kwargs.get('method'))

        return handle_response(response, response_data)
```

Connection type - httpapi...(contd.)

Authentication?

1. HTTP Basic
2. Reusable authentication tokens
3. login() & logout()

httpapi authentication: update_auth

```
def update_auth(self, response, response_text):
    """Return per-request auth token.

    The response should be a dictionary that can be plugged into the
    headers of a request. The default implementation uses cookie data.
    If no authentication data is found, return None
    """
    cookie = response.info().get('Set-Cookie')
    if cookie:
        return {'Cookie': cookie}

    return None
```

httpapi authentication: login & logout

```
class HttpApi(HttpApiBase):
    def login(self, username, password):
        if username and password:
            payload = {'user': username, 'password': password}
            url = '/web_api/login'
            response, response_data = self.send_request(url, payload)
        else:
            raise AnsibleConnectionFailure('Username and password are required for login')

        try:
            self.connection._auth = {'X-chkp-sid': response_data['sid']}
            self.connection._session_uid = response_data['uid']
        except KeyError:
            raise ConnectionError(
                'Server returned response without token info during connection authentication: %s' % response)

    def logout(self):
        url = '/web_api/logout'

        response, dummy = self.send_request(url, None)
```

httpapi and HTTPErrors

```
def handle_httperror(self, exc):
    """Overridable method for dealing with HTTP codes...."""
    if exc.code == 401:
        if self.connection._auth:
            # Stored auth appears to be invalid, clear and retry
            self.connection._auth = None
            self.login(self.connection.get_option('remote_user'), self.connection.get_option('password'))
            return True
        else:
            # Unauthorized and there's no token. Return an error
            return False

    return exc
```

Connection type - httpapi...(contd.)

```
class HttpApiBase(AnsiblePlugin):
    def __init__(self, connection):...

    def set_become(self, become_context):...

    def login(self, username, password):...

    def logout(self):...

    def update_auth(self, response, response_text):...

    def handle_httperror(self, exc):...

    @abstractmethod
    def send_request(self, data, **message_kwargs):...
```

Connection type - network_cli

1) network_cli (plugins/connection/network_cli.py) connection type uses paramiko_ssh under the hood which creates a pseudo terminal to send command and receive response.

2) network_cli loads two platform specific plugins based on the value of **“ansible_network_os”**
   a) Terminal plugin (eg. plugins/terminal/ios.py)
   b) Cliconf plugin (eg. plugins/cliconf/ios.py)

3) The terminal plugin controls the parameters related to terminal like setting terminal length and width, page disabling and privilege escalation. Also it define regex to identify command prompt and error prompts.

4) The cliconf plugin provides an abstraction layer for low level send, receive operations. For eg. edit_config() method ensures the prompt is in config mode before executing configuration commands.

Connection type - network_cli...(contd.)

```
# network_cli connection plugin

class Connection(NetworkConnectionBase):
    ''' CLI (shell) SSH connections on Paramiko '''

    transport = 'network_cli'

    def __init__(self, play_context, new_stdin, *args, **kwargs):...

    def _connect(self):...

    def close(self):...

    @ensure_connect
    def send(self, command, prompt=None, answer=None, newline=True, sendonly=False, prompt_retry_check=False, check_all=False):.

    def receive(self, command=None, prompts=None, answer=None, newline=True, prompt_retry_check=False, check_all=False):...

    @ensure_connect
    def get_prompt(self):...

    def exec_command(self, cmd, in_data=None, sudoable=True):...
```

Connection type - network_cli...(contd.)

```
# plugins/connection/__init__.py

class NetworkConnectionBase(ConnectionBase):
    """
    A base class for network-style connections.
    """
    force_persistence = True
    # Do not use _remote_is_local in other connections
    _remote_is_local = True

    def __init__(self, play_context, new_stdin, *args, **kwargs):...

    def __getattr__(self, name):...
```

Connection type - network_cli...(contd.)

```
# plugins/terminal/ios.py

class TerminalModule(TerminalBase):

    terminal_stdout_re = [...]

    terminal_stderr_re = [...]

    def on_open_shell(self):...

    def on_become(self, passwd=None):...

    def on_unbecome(self):...
```

Connection type - network_cli...(contd.)

```
# plugins/cliconf/ios.py

from ansible.plugins.cliconf import CliconfBase, enable_mode

class Cliconf(CliconfBase):

    @enable_mode
    def get_config(self, source='running', flags=None, format=None):...

    @enable_mode
    def edit_config(self, candidate=None, commit=True, replace=None, comment=None):...

    def get_diff(self, candidate=None, running=None, diff_match='line', diff_ignore_lines=None, path=None, diff_replace='line'):

    def get(self, command=None, prompt=None, answer=None, sendonly=False, output=None, newline=True, check_all=False):...

    def get_device_info(self):...

    def get_capabilities(self):...

    def run_commands(self, commands=None, check_rc=True):...
```

Connection type - network_cli...(contd).

1) To support for new network operating system to work with network_cli connection implement the cliconf and terminal plugins.

2) The plugins can reside in
   a) Adjacent to playbook in folders
      i) “cliconf_plugins/”
      ii) “terminal_plugins/”
   b) Roles
      i) “myrole/cliconf_plugins/”
      ii) “myrole/terminal_plugins/”
   c) Collections
      i) “myorg/mycollection/plugins/terminal/”
      ii) “myorg/mycollection/plugins/cliconf/”

3) After that use ***cli_command*** and ***cli_config*** modules manage the new network os.

Connection type - netconf

1) netconf connection uses ncclient python library under the hood to initiate a netconf session with netconf enable remote network device and execute netconf RPC request and receive response.

2) If the network device support standard netconf (RFC 6241) operation like “get”, “get-config”, “edit-config” etc set the value of **“ansible_network_os”** to **“default”**.

3) “netconf_get”, “netconf_config” and “netconf_rpc” modules can be used to talk to netconf enable remote host.

4) To support vendor specific netconf RPC’s add the implementation in network os specific netconf plugin. For example in case of junos the proprietary RPC methods are implemented in ***“plugins/netconf/junos.py”*** and value of **“ansible_network_os”** is set the name of the netconf plugin file, that is ***“junos”*** in this case.

Connection type - netconf...(contd.)

```
# plugins/netconf/__init__.py

from ansible.plugins import AnsiblePlugin

class NetconfBase(AnsiblePlugin):

    def get_config(self, source=None, filter=None):...

    def get(self, filter=None, with_defaults=None):...

    def edit_config(self, config=None, format='xml', target='candidate',
                    default_operation=None, test_option=None, error_option=None):...

    def validate(self, source='candidate'):...

    def copy_config(self, source, target):...

    def lock(self, target="candidate"):...

    def unlock(self, target="candidate"):...

    def discard_changes(self):...

    def commit(self, confirmed=False, timeout=None, persist=None):...

    def get_schema(self, identifier=None, version=None, format=None):...

    def delete_config(self, target):...

    def get_capabilities(self):...
```

Connection type - netconf...(contd.)

```
# plugins/netconf/junos.py

from ansible.plugins.netconf import NetconfBase

class Netconf(NetconfBase):

    def load_configuration(self, format='xml', action='merge',
                           target='candidate', config=None):...

    def compare_configuration(self, rollback=0):...

    def reboot(self):...
```

#ANSIBLEFEST2019

THANK YOU


youtube.com/AnsibleAutomation


facebook.com/ansibleautomation


linkedin.com/company/Red-Hat


twitter.com/ansible