XC™ Series Ansible Play Writing Guide (CLE 6.0.UP03) S-2582

Contents

1 About the XC™ Series Ansible Play Writing Guide

The XC™ Series Ansible Play Writing Guide is intended to help users of CLE 6.x software extend or customize configuration of nodes to fit their purpose. The CLE 6.x releases use a widely-known system configuration tool called Ansible. This document is not intended to introduce general Ansible use, only to explain how CLE 6.x uses Ansible since the approach is different from traditional Ansible use. Documentation for Ansible is available from *http://docs.ansible.com*. For a comprehensive list of commonly used Ansible terms and their definitions, visit *http://docs.ansible.com/ansible/glossary.html*

Ansible is usually used in "push" mode, where a fully booted driver node communicates with a booted, but unconfigured, set of "target" nodes and communicates a set of actions to be taken to configure services. Since Cray uses Ansible to configure almost all aspects of the system, including network NICs and requires concurrent action on many thousands of nodes, Ansible is used in a less commonly found "pull" mode. Plays run ubiquitously and use provided data to determine what action, if any, is required on the node they are executing on.

Release CLE 6.0.UP03

This publication of XC™ Series Ansible Play Writing Guide supports Cray software release CLE 6.0.UP03, released on 16 FEB 2017

This is the first publication of this document.

Scope and Audience

This publication is intended for system installers, administrators, and anyone who configures software services on a Cray system running SMW 8.0/CLE 6.0. Use of the term *user* throughout refers to the intended audience, not to end users of the system.

Command Prompt Conventions

Host name and account in command prompts

The host name in a command prompt indicates where the command must be run. The account that must run the command is also indicated in the prompt.

- The `root` or super-user account always has the `#` character at the end of the prompt.
- Any non-`root` account is indicated with *`account@hostname>`*. A user account that is neither `root` nor `crayadm` is referred to as `user`.

`smw#`	Run the command on the SMW as `root`.
`cmc#`	Run the command on the CMC as `root`.
`sdb#`	Run the command on the SDB node as `root`.

`crayadm@boot>`	Run the command on the boot node as the `crayadm` user.
`user@login>`	Run the command on any login node as any non-`root` user.
`hostname``#`	Run the command on the specified system as `root`.
`user@`*`hostname`*`>`	Run the command on the specified system as any non-`root` user.
`smw1#` `smw2#`	For a system configured with the SMW failover feature there are two SMWs—one in an active role and the other in a passive role. The SMW that is active at the start of a procedure is *smw1*. The SMW that is passive is *smw2*.
`smwactive#` `smwpassive#`	In some scenarios, the active SMW is *smw1* at the start of a procedure—then the procedure requires a failover to the other SMW. In this case, the documentation will continue to refer to the formerly active SMW as *smw1*, even though *smw2* is now the active SMW. If further clarification is needed in a procedure, the active SMW will be called *smwactive* and the passive SMW will be called *smwpassive*.

Command prompt inside chroot

If the `chroot` command is used, the prompt changes to indicate that it is inside a chroot environment on the system.

```
smw# chroot /path/to/chroot
chroot-smw#
```

Directory path in command prompt

Example prompts do not include the directory path, because long paths can reduce the clarity of examples. Most of the time, the command can be executed from any directory. When it matters which directory the command is invoked within, the `cd` command is used to change into the directory, and the directory is referenced with a period (`.`) to indicate the current directory.

For example, here are actual prompts as they appear on the system:

```
smw:~ # cd /etc
smw:/etc# cd /var/tmp
smw:/var/tmp# ls ./file
smw:/var/tmp# su - crayadm
crayadm@smw:~> cd /usr/bin
crayadm@smw:/usr/bin> ./command
```

And here are the same prompts as they appear in this publication:

```
smw# cd /etc
smw# cd /var/tmp
smw# ls ./file
smw# su - crayadm
crayadm@smw> cd /usr/bin
crayadm@smw> ./command
```

Feedback

Visit the Cray Publications Portal at *http://pubs.cray.com* and make comments online using the **Contact Us** button in the upper-right corner or Email pubs@cray.com. Your comments are important to us and we will respond within 24 hours.

Typographic Conventions

`Monospace`	Indicates program code, reserved words, library functions, command-line prompts, screen output, file/path names, and other software constructs.
`Monospaced Bold`	Indicates commands that must be entered on a command line or in response to an interactive prompt.
`Oblique` or *`Italics`*	Indicates user-supplied values in commands or syntax definitions.
Proportional Bold	Indicates a **GUI Window**, **GUI element**, cascading menu (**Ctrl**→**Alt**→**Delete**), or key strokes (press **Enter**).
\ (backslash)	At the end of a command line, indicates the Linux® shell line continuation character (lines joined by a backslash are parsed as a single line).

Trademarks

2 An Overview of Ansible on a Cray System

Separating Content from Configuration

The approach to system configuration in CLE 6.x releases is very different to previous Cray releases. The intention is to separate the node configuration data from the boot image propagated at boot time. This allows images to be built once and then assigned to nodes at boot time, or even transferred to other Cray 6.x systems and booted against configuration data on that system. To make boot images portable between nodes on the HSN, they contain very little configuration. Instead, each node configures required services and functionality on every boot, driven by configuration data stored on the SMW, a "config set", that is propagated to all nodes comprising the system. This data allows Ansible plays to make coherent changes to configuration within the booting image to provide system services across the Cray system.

2.1 Determine When Plays Are Run

As CLE nodes boot, they run the script `/init` which provides low-level initialization. Under pre-CLE 6.x releases, the script would identify the node's service/compute type and load kernel modules appropriate for the environment. The script would then invoke the Linux `init(1)` process and system services would be started as codified in the shared root filesystem. This was assumed to proceed without error and leave the node in a usable state.

In CLE 6.x releases, configuration is performed by Ansible plays and `/init` uses Ansible to complete most configuration tasks. Early `/init` processing initializes the available NICs based on kernel command line parameters and makes config set data available. Ansible plays are then run, before the `systemd(1)` system is started. This allows time for plays to control loading of modules, or enabling of operating system services during later `systemd(1)` startup processing. Note that `init(1)` is still used for DAL nodes running CentOS 6.5; however, since most nodes run SLES 12, `systemd(1)` should be preferred over `init(1)`.

When control is transfered to `systemd(1)`, "normal" Linux startup occurs and `systemd` starts services in much the same way as previous CLE releases. When `Run Level 3` or `Multi.User.target` is reached, a final run of Ansible plays is made. This final run starts services and facilities that rely on standard Linux services started by `systemd(1)`, or provide higher level coordination with other nodes. Late in this sequence of plays, `NodeHealthCheck` is invoked and, if successful, it generates an HSS event that marks the node as booted. This is intended to provide a clear indication that not only has `systemd(1)` completed, but that the set of services the config set specifies for this node have been properly initialized.

2.2 Determine Which Plays Are Run

While boot images have little configuration at boot time, the images do contain all Cray-provided plays for any functionality they contain. The plays are installed in the directory `/etc/ansible` along with other Ansible RPM content. These plays are supplemented by plays found in the `ansible` directory of the config set with which the booting node is associated.

The `cray-ansible` script, found in `/etc/init.d`, is used as a wrapper to start Ansible plays. The script looks for files suffixed with `.yaml` in `/etc/ansible` and in the `ansible` directories of the current and global config sets. Thus plays specific to the system, or even the config set, are treated as peers of Cray provided plays during the boot process.

Plays can be optionally designated as arbitrary *types* and cray-ansible can be directed to run plays of specific types. The default *type* for plays not explicitly declaring such can also be designated when cray-ansible runs. This play type selection is used, for instance, during `/init` on nodes designated as `netroot` nodes to mount the root filesystem image from the boot node.

2.3 The Order in Which Ansible Plays Are Run

When `cray-ansible` has selected a series of plays for execution it produces a file that includes each candidate play and calls `ansible-playbook(1)` to run that file. The order that plays occur in the file determines the order of operation. This has implications for services with dependencies on other services and functionalities. Cray-provided plays signal to `cray-ansible` their ordering requirements using a convention based on play variables.

The `cray-ansible` script will group plays into three general groups: plays that declare or provide dependencies to other plays, plays that should be run late, and plays that don't indicate or provide dependenies for another play. Plays that don't declare the *`run_late`* variable but do declare or provide dependencies to other plays run first, while plays that want to run late declare the variable `run_late`. Plays that do not declare *`run_late`* or *`run_after`* and are not referenced in any *`run_after`* list are run in indeterminate order between the other two groups. When developing plays, identify other plays that may be modifying the same file and set up play dependency accordingly.

Plays provided by Cray will use a variable called `run_after` to list any plays that should run before the current play is run. The plays listed by `run_after` are not required to exist; non-existent plays are ignored, but all existing plays in the list will run before the play declaring the *`run_after`* variable. The list of plays to run behind does not require a path element or a `.yaml` suffix. The following declaration demonstrates a list of plays to run beforehand:

```
=
            vars:
            - run_after:
                - common
                - task1
=
```

The above declaration specifies that the play should not run before `/etc/ansible/common.yaml` or `/etc/opt/cray/config/current/ansible/task1.yaml` if it exists.

In order to allow config set provided plays to insert themselves into the ordering of Cray supplied plays, it is also possible to declare a similar variable called `run_before`. This variable effectively adds the nominating play's name to the `run_after` list in the listed plays.

NOTE: The ordering of plays is not affected by whether the plays are running before or after `systemd` start of system services.

2.4 Data Available to Plays

There are two broad categories of data that drive configuration: the config sets provided by IMPS Distribution Service (IDS), and Ansible "facts" provided to all plays by Ansible infrastructure. Both categories appear to plays as hierarchy's of dotted variables of values, lists and dictionaries. For instance, the fact `ansible_devices.sda.size` gives the size of sda on the node the play is running on as a string, whereas `cray_node_groups.settings.groups.data` is the list of defined node groups for the config set associated with the node.

The set of configuration plays selected by `cray-ansible` are commonly the same across large numbers of the nodes comrpising the system. All nodes booting the same computer node image will, for instance, run exactly the same set of plays. Similarly, all service node images will result in the same set of plays running during boot on every service node. Differences in configuration of the individual nodes comes from how those plays react to the data they consume.

2.4.1 Ansible Facts

More information on Ansible facts is available from the standard Ansible documentation, but it is worth noting that Cray uses the provided framework to make facts available to plays based on the running node's configuration. All such facts appear under `ansible_local.cray_system`.

Table 1. Commonly Used Cray Ansible Facts

Name	Description
`is_cray_blade`	A boolean value. The node is on the HSN if this value is `true`.
`hostid`	A string. The cname of HSN nodes, or the data returned by `hostid(1)` for elogin or SMW instances.
`node_groups`	A list of strings. The node groups that the current node is a member of.
`in_init`	Whether the play is currently running before system services are available.

The fact `ansible_local.cray_system.in_init` can be used to separate actions taken to configure things that normal Linux boot processing should do, including things that require system services to be started. A complete list of current Cray supplied facts can be seen in the output of the script `/etc/ansible/facts.d/cray_system.fact` as seen below on an SMW node:

```
smw# /etc/ansible/facts.d/cray_system.fact
{"hostid": "1eac3f0c", "roles": ["smw"], "platform": "unknown",
"is_cray_blade": false, "host_type": "management", "uses_systemd": true,
"in_init": false, "sessionid": "", "node_groups": [], "nims_group": [],
"standby_node": false}
```

The same script running on a compute node:

```
nid00023#  /etc/ansible/facts.d/cray_system.fact
{"topology_class": 2, "mcdram_cfg": "", "primary_boot_node_cname": "c0-1c0s0n1",
"node_groups": ["login_nodes", "service_nodes", "all_nodes"], "backup_boot_node":
145, "sdb_node": 386,
"primary_sdb_node_cname": "c0-1c0s0n2", "platform": "service",
"max_torus_dimension": [1, 5, 15],
"max_node_id": 767, "nid": 13, "in_init": false, "sessionid":
"p0-20170213t101658", "hostid": "c0-0c0s3n1",
"standby_node": false, "num_sys_nodes": 158, "max_sys_nodes": 768, "roles": [],
"is_cray_blade": true,
"uses_systemd": true, "cname": "c0-0c0s3n1", "sys_nodes": [1, 2, 5, 6, 9, 10, 13,
14, 17, 18, 20, 21, 22, 23,
24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43,
44, 45, 46, 47, 48, 49, 50,
51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 65, 66, 68, 69, 70, 71, 73,
74, 77, 78, 80, 81, 82, 83,
89, 90, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107,
120, 121, 122, 123, 124, 125,
126, 127, 128, 129, 130, 131, 133, 134, 136, 137, 138, 139, 140, 141, 142, 143,
145, 146, 148, 149, 150, 151,
152, 153, 154, 155, 156, 157, 158, 159, 168, 169, 170, 171, 172, 173, 174, 175,
176, 177, 178, 179, 384, 385,
386, 387, 388, 389, 390, 391, 392, 393, 394, 395, 400, 401, 402, 403, 416, 417,
418, 419, 424, 425, 426, 427],
"host_type": "", "boot_node": 385, "nims_group": ["login"], "backup_sdb_node": 74}
```

2.4.2 Config Set

Config set data is comprised of two parts: the global config set and the CLE config set. The global config set is common to the SMW and all partitions. The CLE config set is associated with a CLE partition and passed to a booting CLE node as a kernel parameter which is prepared using NIMS (cnode command). The eLogin nodes also have both the global and CLE config sets available to them. On the SMW, the config set data is located in these directories:

- /var/opt/cray/imps/config/sets/global
- /var/opt/cray/imps/config/sets/**CLE_config_set_name**

On a node, the config set data is located in these directories:

- /etc/opt/cray/config/global
- /etc/opt/cray/config/current

NOTE: Notice that this is the current CLE config set for this node and is always called current.

Within each of these directories are subdirectories named config which house the actual YAML files that contain data to be used by plays. If the files end with `_config.yaml`, the contents are combinations of configured values and metadata used by the `cfgset(8)` utility to maintain the YAML files. Each file introduces a namespace that will be available to plays at runtime when cray-ansible is used to prepare for play execution; for example, `cray_net_config.yaml` provides `cray_net`.

Generally, the current and global config set data sets use different namespaces for configuration values. However, some configuration data might logically not depend on which config set is actually current, resulting in configuration data using both namespaces. For example, the config data file `cray_time_config.yaml` exists in both current and global directory trees; both current and global data sets provide `cray_time`. This conflict is resolved using the `inherit` Boolean value setting. The `inherit` setting determines whether to defer to the

global settings when a conflict is detected. If `inherit` is set to true, values are inherited from the global config set; otherwise, if no conflict is detected, the current specified values are used.

Files in config set directories can also provide non-configurator data. If a file ends with the `.yaml` suffix but lacks `_config` in its name, any correctly formatted YAML structures will be available to plays using the config set. The files will be ignored by the configurator and can be used for manually maintained configuration settings for ad-hoc use. Since the runtime behaviour of plays without `_config` is undefined, care should be taken to avoid duplicately-named YAML files that are not designated with `_config` in the filename, or commonly named data structures, in the `global` and `current` config subdirectories.

2.5 Audit Trail of Actions

When cray-ansible runs Ansible plays, log data is captured in files in the directory `/var/opt/cray/log/ansible`. Generally, if errors occur during boot for a CLE node or an eLogin node, cray-ansible will attempt to copy the tail of these log files to the console device in case the node is unresponsive to login attempts. Each time cray-ansible runs, it rotates these logs; however, the number of logs kept is limited. If Ansible runs are restarted, some history may be lost. For the SMW, the log files are:

- /var/opt/cray/log/ansible/ansible-booted
- /var/opt/cray/log/ansible/file-changelog-booted
- /var/opt/cray/log/ansible/file-changelog-booted.yaml

For CLE nodes which are not booted with netroot images and for eLogin nodes, the log files from when `/init` calls cray-ansible in the init phase are:

- /var/opt/cray/log/ansible/sitelog-init
- /var/opt/cray/log/ansible/file-changelog-init
- /var/opt/cray/log/ansible/file-changelog-init.yaml

For CLE nodes which are not booted with netroot images and for eLogin nodes, the log files from when `systemd` calls cray-ansible in the booted phase are:

- /var/opt/cray/log/ansible/sitelog-booted
- /var/opt/cray/log/ansible/file-changelog-booted
- /var/opt/cray/log/ansible/file-changelog-booted.yaml

For CLE nodes which are booted with netroot images, the log files from when `/init` calls cray-ansible in the netroot setup phase are:

- /var/opt/cray/log/ansible/sitelog-init-netroot_setup
- /var/opt/cray/log/ansible/file-changelog-init-netroot_setup
- /var/opt/cray/log/ansible/file-changelog-init-netroot_setup.yaml

For CLE nodes which are booted with netroot images, the log files from when `/init` calls calls cray-ansible in the init phase:

- /var/opt/cray/log/ansible/sitelog-init
- /var/opt/cray/log/ansible/file-changelog-init
- /var/opt/cray/log/ansible/file-changelog-init.yaml

For CLE nodes which are booted with netroot images, the log files from when `systemd` calls cray-ansible in the booted phase are:

- /var/opt/cray/log/ansible/sitelog-booted
- /var/opt/cray/log/ansible/file-changelog-booted
- /var/opt/cray/log/ansible/file-changelog-booted.yaml

For details about cray-ansible being run during the booting process, see the XC™ Series Boot Troubleshooting Guide.

2.6 Simple Shares

The Simple Filesystem Sharing service quickly shares files between compute nodes that are connected to the high speed network (HSN). Plays that declare the variable *`fs_export_needed`* and either include the *`fs_share`* role, or make it a dependency of the play, expect the named directory to appear in the `cray_simple_shares` configuration file. That storage will be mounted and shared across CLE nodes as specified in the simple shares config before the play runs. Simple shares are not meant to cover all cases of intra-node mounting, but have several convenient use cases. Any play can use this mechanism, as several Cray provided plays do.

3 An Approach to Play Development

Ansible's flexible nature allows many different approaches to the development of Ansible plays. Presented here are Ansible play creation suggestions and guidelines based on what Cray has found to be desirable properties for configuring XC Series systems. There are several conventions that Cray-provided plays adhere to that are not requirements forced by Ansible. Initially, avoiding these Cray-provided conventions means that functional plays can be developed and distributed across a system without taking part in boot time configuration. Even when plays are placed in their final locations and file names, as previously noted, they can declare `cray_play_types` which will avoid their invocation until confidence is built in their operation. While adding plays to boot time configuration is not difficult, the fact that all boot time configuration is expected to succeed to allow predictable final state means that a minor mistake can have major impacts on the ability to reboot systems and return them to a productive state. The use of NIMS maps and Config Set clones can be used to limit the impact of boot time problems to specific, predictable, nodes which might allow normal operation on the majority of nodes as development continues. It is important to validate play behavior before committing the entire system to their utility. The steps outlined in this section help to avoid testing plays via system reboots.

NOTE:

The following changes are introduced in UP03 and should be noted by customers upgrading from UP01 or UP02 installs and site local plays.There are only two major Ansible play changes with UP03. These changes may adversely affect existing site-local plays:

- `xc_node` top level play has been removed
- `cle_lustre_client` top level play is now `cray_lustre_client`

All existing top level plays that have been previously configured still exist and function as no-op checkpoints for their prior functionality. Checkpoints will be deprecated in future releases; determining proper *`run_after`* order at this time is strongly suggested. Top level plays scheduled for future deprecation are:

- `early`
- `compute_node`
- `service_node`
- `common`
- `login_node`

3.1 Syntax Checking and Prototyping

Since configuration plays are ubiquitous and driven by data, it is often useful to investigate the data which will be used to control the actions of plays. Aside from reading Cray plays or files in the config directory, there are a number of ways to explore the data available to plays.

Ansible The `ansible` command can provide a comprehensive list of facts available to plays at runtime using the "setup" module. The list of facts available is extensive and includes values from all plugins. The following command line example demonstrates using the `ansible` command to lookup the value for the `ansible_kernel` fact on a boot node and smw respectively.

```
boot# ansible -m setup localhost | grep ansible_kernel
            "ansible_kernel": "3.12.60-52.49.1_2.2-cray_ari_s"
```

```
smw# ansible -m setup localhost | grep ansible_kernel
           "ansible_kernel": "3.12.51-52.39-default"
```

Fact Plugins Facts can be augmented by adding files or scripts to installations in drop directories. CLE, SMW, and eLogin nodes will provide different sets of facts using this mechanism, which must be accommodated by plays installed in all three environments. These files are easy sources to examine for Cray-specific facts. The example below demonstrates running the `cray_system.fact` script on an eLogin node to view a list of available Cray system facts. For more information on Ansible facts, see section *Ansible Facts* on page 8.

```
elogin# /etc/ansible/facts.d/cray_system.fact
            {"hostid": "percival-elogin1", "roles": [], "platform": "unknown",
            "is_cray_blade": false, "host_type": "", "uses_systemd": true,
"in_init": false,
            "sessionid": "", "node_groups": ["elogin_nodes"],
            "nims_group": [], "standby_node": false}
```

cfgset search The configurator `cfgset` command features several subcommands that are helpful for exploring system data. The example below shows searching all level `advanced` data in service `cray_global_net` for the term `admin global`.

```
smw# cfgset search --level advanced -s cray_global_net -t admin global | head -15

            INFO - Checking services for valid YAML syntax
            INFO - Checking services for schema compliance
            # 23 matches for 'admin' from cray_global_net_config.yaml

#-------------------------------------------------------------------------------
            cray_global_net.settings.networks.data.admin.description: Network
that connects the SMW, boot and SDB nodes.
            cray_global_net.settings.networks.data.admin.ipv4_network: 10.3.0.0
            cray_global_net.settings.networks.data.admin.ipv4_netmask:
255.255.0.0
            cray_global_net.settings.networks.data.admin.ipv4_broadcast: #
(empty)
            cray_global_net.settings.networks.data.admin.ipv4_gateway: # (empty)
            cray_global_net.settings.networks.data.admin.dns_servers: [ ] #
(empty)
            cray_global_net.settings.networks.data.admin.dns_search: [ ] #
(empty)
            cray_global_net.settings.networks.data.admin.ntp_servers: [ ] #
(empty)
            cray_global_net.settings.networks.data.admin.fw_external: false

cray_global_net.settings.hosts.data.primary_smw.interfaces.admin_interface.name:
eth3
```

For a full list of `cfgset` commands, as well as examples of how to use them, see the *XC_Series_Configurator_User_Guide_CLE60UP03_S-2560.ditamap#C320271*.

cfgsetquery This tool searches config set data on SMW or HSN nodes. `cfgsetquery` only searches for variable name and sub path matches, but provides the namespace path, helpful when writing Ansible plays. This command can be run anywhere config set data is found, including eLogin nodes. The example below demonstrates searching for all config set values that contain the `networks.data` path.

```
smw# /opt/cray/cfgutils/bin/cfgsetquery networks.data | head -11
                  cray_global_net.settings.networks.data:
                  -     ipv4_gateway:
                         dns_servers:
                         description: Network that connects the SMW,
boot and SDB nodes.
                         fw_external: False
                         ntp_servers:
                          dns_search:
                        ipv4_network: 10.3.0.0
                        ipv4_netmask: 255.255.0.0
                                 key: admin
                      ipv4_broadcast:
```

Configurator `Multival` Fields

While administrators are provided tools that help isolate them from some details of configuration data, play writers are forced to be aware of some of the configurator data handling details. The most commonly encountered of these details are data dictionaries and the handling of `multival` fields. `Multival` fields are lists of collections of config values that administrators might want to refer to logically.

In the configurator output displayed in the `cfgset search` example, the output shows the namespace path to the `admin` data to be `cray_global_net.settings.networks.data.admin`. In YAML format, this path is a set of dictionaries with nested keys for each of the elements. An administrator may want to logically refer to the group of values, `admin`, by a name that is not a valid YAML identifier. To avoid this, the configurator stores the collections of values as a list and adds a `key` field which contains the `admin` element's logical name. It then converts between the stored and the displayed presentations as needed. Since the configurator keeps metadata-describing fields in the same file as the fields themselves, the configurator maintains separation between the two by storing values in a dictionary named `data` and metadata in a dictionary named `configurator`. These dictionaries are apparent when examining the config files, but are hidden in configuration presentation to administrators.

In contrast, the output of `cfgsetquery` only hides the configurator metadata. The paths presented can be used directly in plays as variable references and the "multival" value containing the values for a network are exposed as a list, with the key value. Processing the data in a play will need to walk the list and match keys.

3.2 Pull-mode Boilerplate

On Cray systems, Ansible is used in pull mode. Most Cray provided plays take the same form. Plays are comprised of two parts: a top-level file that declares variables (for example, declaring play dependencies), and

roles. The roles of a play are used to implement changes, often with `when` clauses in the case of `cray_managed` services. Most Cray provided templates and config files have a hidden setting which allows customers to indicate they will manage the service or functionality themselves. Note that this setting is separate from the `enabled` boolean flag which might be used to shut down a running service if set to `false`.

The following boilerplate from Cray-provided plays can be used to model new plays:

```
- hosts: localhost
  name: "Network's status"
  vars:
  - run_late: True
  tasks:
  - name: "Show known networks' names"
    debug: var=item.key
    with_items: cray_global_net.settings.networks.data
  - name: "Ask systemctl about network service"
    shell: systemctl status -l network | head -5
    when: not ansible_local.cray_system.in_init
    register: systemctl
  - name: "Show what we learned"
    debug: var=systemctl.stdout
```

A play is a yaml list of dictionaries containing one element, with keys *`hosts`*, *`name`*, *`vars`*, and *`tasks`*. The main difference between this example play and a Cray provided "top level" play, is that this play uses tasks, rather than roles, to perform actions. The list of hosts that are targeted is always *`localhost`*; note that this is a pull mode, self initiated play. The list element's *`name`* key is used to find the name that should be used in any logging generated by use of the play. The tasks are similarly tagged and will also be identified by those strings in logs. This is a useful practice to maintain as it makes analysis of activity easier in debug. Explicit use of quotes for the name string allows characters to be used that would otherwise cause YAML parsing errors. While it is unused here, the *`run_late`* variable will place this play towards the end of cray-ansible activity if it is renamed or copied into one of the well known locations mentioned previously. Finally, one of the tasks uses systemctl to check on the state of a service, this will only work after systemd starts, so that task is constrained to act only if the available facts indicate that the play is not acting while in the `/init` script. This boilerplate can be placed in any convenient file during development. Placing it in a file such as `/root/ns` eliminates the possibility of interference with normal processing when run to test its behavior:

```
boot# ansible-playbook /root/ns
...
PLAY [Network's status] ********************************************************

GATHERING FACTS ****************************************************************
ok: [localhost]

TASK: [Show known networks' names] *********************************************
ok: [localhost] => (item={'ipv4_gateway': '', 'dns_servers': [], 'description':
'Network that connects the SMW, boot and SDB nodes.', 'fw_external': False,
'ntp_servers': [], 'dns_search': [], 'ipv4_network': '10.3.0.0', 'ipv4_netmask':
'255.255.0.0', 'key': 'admin', 'ipv4_broadcast': ''}) => {
    "item": {
        "description": "Network that connects the SMW, boot and SDB nodes.",
        "dns_search": [],
        "dns_servers": [],
        "fw_external": false,
        "ipv4_broadcast": "",
        "ipv4_gateway": "",
        "ipv4_netmask": "255.255.0.0",
        "ipv4_network": "10.3.0.0",
        "key": "admin",
```

```
        "ntp_servers": []
    },
    "var": {
        "item.key": "admin"
    }
}
...
        "ipv4_network": "172.30.12.0",
        "key": "management",
        "ntp_servers": [
            "cfntp-4-1",
            "cfntp-4-2"
        ]
    },
    "var": {
        "item.key": "management"
    }
}

TASK: [Ask systemctl about network service] ***********************************
changed: [localhost]

TASK: [Show what we learned] **************************************************
ok: [localhost] => {
    "var": {
        "systemctl.stdout": "wicked.service - wicked managed network interfaces
\n   Loaded: loaded (/usr/lib/systemd/system/wicked.service; enabled)\n   Active:
active (exited) since Thu 2016-11-17 13:24:23 CST; 51min ago\n Main PID: 6033
(code=exited, status=0/SUCCESS)\n   CGroup: /system.slice/wicked.service"
    }
}

PLAY RECAP ********************************************************************
localhost                  : ok=4    changed=1    unreachable=0    failed=0
#
```

Note that much of the output is not shown in the above screenshot, but the various names and processing elements from the networks should be apparent. There's nothing inherently wrong with a simple play having a few tasks in a top level file as shown here, but there are some drawbacks. For instance, if the play was added as-is to the boot-time configuration, then the list processing would occur twice, as would the *`Show what we learned`* task. That's not a particular problem in this instance - if we wished to change the behaviour we could add *`when:`* to the two tasks and replicate the conditionals. But that does open the door to simple human error if the conditions change, perhaps as the nodes targeted to take the actions changed during devlopmental testing.

The above play can be re-arranged and made more convenient by using a role and adding conditions when invoked. When a play references a role, Ansible will look for the specified role in three different locations: in a `roles` directory co-located with the play, in `/etc/ansible`, and in the config set. During testing, creating a role directory as a peer of the play being tested allows roles to be associated with the developing play. Because of the common list of directories searched, role names should be carefully considered to avoid possible confusion with existing Cray-written role names. Locally devised roles can be prefaced with `org_` (or another chosen prefix) for easy identification. If the example of the tasks above were placed in the file `roles/sle_net_serv_chk/tasks/main.yaml`, the top level play might become:

```
- hosts: localhost
  name: "Network's status"
  vars:
  - run_late: True
  roles:
```

```
  - role: sle_net_srv_chk
    when: not ansible_local.cray_system.in_init
```

The effect of the *when* clause on either role invocation or *include* statements is to add an implicit matching *when* clause to every task in the referenced text, and end with *when* clauses that those tasks specify (in this example, the three tasks previously in the top level play). This makes the *when* clause guarding the `systemctl` use redundant and allows other constraints to be later applied to all tasks in the role by extending the *when* clause in the top level play. During play development, this is a convenient way of affecting where and when the play takes effect.

3.3 Using a Config Set to Distribute an Ansible Play to All Nodes on a System

The tools and techniques listed here can assist with integrating an Ansible play into a config set. Working with files in tmpfs storage limits both the longevity of the play and its utility across the system. The use of user or admin accounts on a system allows use of persistent storage and can be useful for testing on a wider set of nodes if the directories are widely cross-mounted. If the play being developed is configuring facilities not currently covered by Cray-provided plays, it is possible to develop the play in isolation - timing of the integration of the play into the overall boot-time configuration runs becomes a matter of convenience. If the play provides a modified environment for Cray plays, or has dependencies on modified configurations from Cray plays, it is often necessary to perform play integration with the config set before play development completes. In these cases, it is recommended to control what the play under development uses and where the play under development is used. The tools and techniques listed here can assist with integrating an Ansible play into a config set.

Use `Cray-Ansible Link` to Refresh Config Set Data

Content in the the config set is cached on each node in the system. The CLE nodes use a squashfs image to reduce memory impact of config set data on nodes. Previously, cray-ansible's linking of config set data into the environment for plays was mentioned. At the time that linking is done, cray-ansible also checks to see that the most recent version of config set data is in use on the node. If a play is being developed, it's possible that the config set will be modified in order to change the play or the data it operates on. In such cases it will be necessary to invoke the `link` command. The `link` command in this instance is used to check for new config set data. If new config set data is available, it is pulled to the node and the new config set data is made available. In this first example, new data is available for the p0 config set as shown by the IDS INFO line:

```
hostname# /etc/init.d/cray-ansible link
IDS INFO - Cached squashfs '/var/opt/cray/imps-distribution/source/config/sets/p0'
checksum differs from upstream; replacing local content with upstream
configuration.

Updating host_vars, this may make it harder to interpret sitelogs.
...
```

In this second example, the lack of a notification shows that no new data is available:

```
hostname# /etc/init.d/cray-ansible link

Updating host_vars, this may make it harder to interpret sitelogs.
...
```

Note that the update of the config set is node-local. If the play is being widely tested, it may be necessary to run the `link` command on each node.

Declare Play Types with *cray_play_type*

While early play development allows any file name or location to be used for plays, when testing with other boot-time configuration begins, the standard config set locations and file naming conventions must be followed. Cray-ansible examines plays for various attributes using a convention of variable declarations. One indication a play can make is its *type*; a play can declare itself to be an arbitrary type using the *cray_play_type* variable:

```
  vars:
    cray_play_type:
    - prototype
```

The above fragment shows the play is type *prototype*. The default play type, for plays not declaring an explicit type, can be set when cray-ansible is executed (the list of play_types a cray-ansible run should use can also be set at execution time). In order to maintain isolation of unproven or in-development plays, these plays can indicate that they are of a non-default type; cray-ansible can then be directed to include that type when invoked after boot to test the play. When cray-ansible runs at boot time, it uses a default play type of *cle* and selects plays of type *cle* to run. This means that a play of type *prototype* would not be invoked at boot time, or when cray-ansible is invoked as *normal*:

```
hostname# /etc/init.d/cray-ansible start

cray-ansible: /etc/ansible/site.yaml completed after boot - SUCCEEDED.
Sending ec_node_info with boot code 8 (NODE_INFO_OS_BOOT_SUCCEEDED) for nid 4
hostname# grep -C 2 new.yaml /etc/ansible/site.yaml
- include: /etc/ansible/sysenv.yaml
#Play's play types (prototype) are excluded
#- include: /etc/opt/cray/config/current/ansible/new.yaml
- include: /etc/ansible/wlm_trans.yaml
- include: /etc/ansible/xtremoted.yaml
```

Notice that in the above case, similar to boot node, the new play was excluded - but can easily be included if any run if cray-ansible is run with the appropriate arguments. This allows testing of operations post-rc/systemd start wtihout impacting system boot time operation. The following example shows running cray-ansible and including plays with type *cle* and *prototype*:

```
hostname# /etc/init.d/cray-ansible -t cle,prototype start
cray-ansible: /etc/ansible/site.yaml completed after boot-prototype - SUCCEEDED.
Sending ec_node_info with boot code 8 (NODE_INFO_OS_BOOT_SUCCEEDED) for nid 4
hostname# grep new /etc/ansible/site.yaml
- include: /etc/opt/cray/config/current/ansible/new.yaml
#
```

Test Play Boot Time Behavior

It is necessary to test boot-time behavior of new plays to ensure proper function. Since the config set is being used to propagate both the play in development and possible new data (or modified existing data), it can be helpful to isolate the scope of changes. The following procedure allows a rapid switch between production and play development configurations, useful when circumstances temporarily halt play development.

1. Clone the target config set. The cloned config set will receive the new plays and data:

```
smw# cfgset create --clone p0 p0.proving
```

2. Clone the active NIMS map. The cloned NIMS map will be modified for testing:

```
smw# cmap list | grep -i true
smw# cmap create --clone <grepped-map> p0.proving
```

3. Start Proofing. Making the cloned map active allows config set associations to be changed. The <test-node> will be used for proofing, including node reboot behavior as needed. Nodes can be designated as test nodes by simply changing the config set associated with them:

```
smw# cmap setactive p0.proving
smw# cnode update -c p0.proving <test-node>
```

4. Switch back to production. As circumstances require, the system can be restored to normal operating by updating the active NIMS map:

```
smw# cmap setactive <grepped-map>
smw# xtbootsys --reboot <test-nodes>
```

   Return to play development can be achieved in the same was as production configuration was restored, using the p0.proving map in the last step listed above.

Test Different Node Environments

It is important to test plays not only on nodes intended to be modified, but also on nodes that are not intended play participants since all plays are candidates for inclusion on all nodes if the play *type* is set to *cle* or *undefined*. Similarly, normal boot time activity will run the play both before and after rc/systemd starts. In the p0.proving NIMS test map, it's worth including a node with each image used by the system that is not being affected by new plays being tested: WLM, netroot, elogin, and DAL nodes shouldn't be forgotten if used in the system.

4 Ansible Limitations and Caveats

Service Configuration in `in_init`

One of the reasons cray-ansible runs before systemd starts is to allow plays to influence which standard Linux services will be started during the boot with minimal Ansible action. However, because systemd is not yet started, normal procedures to enable the service via systemctl usually cannot be used. Systemd's current mechanism for recording enabled services' states are filesytem-based and use simple symbolic links. The following is an example of how the standard syslog service is enabled as part of the boot:

```
- name: task llm, enable/disable rsyslogd
  file:
    path: /etc/systemd/system/multi-user.target.wants/rsyslog.service
    state: "{{ 'link' if ansible_local.cray_system.platform != 'compute' else
'absent' }}"
    src: /usr/lib/systemd/system/rsyslog.service
    force: yes
  when: ansible_local.cray_system.in_init
    and ansible_local.cray_system.uses_systemd
```

Notice the guards to ensure that the action is only taken on nodes that use systemd and when running before systemd starts. The task also behaves differently on compute and non compute nodes.

References Parsed Even if Skipped

It is common for roles to use the *`set_fact`* module to update the data available for plays at runtime. This can lead to problems if the fact is referenced in some contexts later. If a contraint is placed on the role that casues the *`set_fact`* to be skipped, and a later task references the fact in a *`when`* clause, for instance, the fact will be undefined and cause the play, and the cray-ansible run, to fail even though the same constraint that skipped the *`set_fact`* will skip the failing task. It is not always easy to tell whether a fact reference will be parsed by Ansible, but in cases where it does occur using the jinja filter *`|default(true)`* will avoid the error by providing a value. Thorough testing on uninvolved nodes will help identify such issues.