Automatically Build, Test and Deploy Your Network Configurations

NANOG 63

Carlos Vicente
Dyn
Project name: Kipper

- Very mellow dog
- Likes an easy, simple life

Can automation do that for us network engineers?
The problems

- Copy-pasting and hand-crafting configurations at the command line:
  - Is very error-prone
  - Leads to inconsistencies
  - Takes too long
  - Does not scale!

- No formal approval process
  - Bad changes can be introduced without review
The goals

● Facilitate consistency
  ○ Use templates to ensure:
    ■ standardization, accuracy and predictability

● Minimize errors
  ○ Avoid direct CLI access
  ○ Formal review/approval process
  ○ Automated tests

● Increase speed
  ○ Handle bulk changes
More goals

- Use open source when possible
  - Build on community efforts
  - Don’t break the bank
  - Well known, well tested

- Use open standards when possible
  - Don’t reinvent the wheel
  - More chances of reusing solution for multiple platforms/vendors
Learn from software engineers

- Continuous Integration and Delivery (CI/CD)
  - Frequent individual integrations into master repository
  - Automated build, test, deploy
  - Identify errors as quickly as possible
  - Many tools already available
    - Git, Jenkins, TravisCI, etc.
Learn from sysadmins

- Many config management tools available
  - Chef, Puppet, Ansible, Salt, etc.
- Domain-specific languages (DSLs) to specify desired state
  - Minimal or no programming required
  - Combined with version control/ distributed workflows
  - Install packages, generate configs, automate checks
What about networks?

- These concepts, processes and tools are still very much missing from network environments
  - Can we use what’s already available?
Dyn case

- ~20 data centers in 5 continents
- Hundreds of network devices
- Multiple teams
- Fast growth
- Automation is a main priority
  - Existing automation of servers and applications
  - Little or no network automation as of last year
NETCONF

- IETF standard for network configuration management
  - RFC 6241
  - Concept of “candidate” configuration
  - XML encoding of data and operations
  - Secure transport (SSH)
- Good support on Juniper
  - Not so on other platforms, unfortunately
Ansible

- Open source IT automation tool
- Focused on simplicity
- Agentless!
  - Uses SSH
- *Push* instead of *pull* model
- Extensible (with modules)
  - Juniper wrote NETCONF module:
    - [https://github.com/Juniper/ansible-junos-stdlib](https://github.com/Juniper/ansible-junos-stdlib)
Github

- Collaboration tool based on Git
- Adds important features
  - Forks and pull requests
  - Useful web interface
  - Gists
  - many more
- Free hosted use for public repos
  - Can do private repos or appliances for a fee
Jenkins

- Popular open source CI/CD tool
  - Many plugins available
- Automates the execution of tasks
  - Cron jobs
  - Events
    - Integrates with external version control
    - Triggers jobs when things change (e.g. pull requests, merges)
Concept
Organization

- **Inventory**
  - All devices grouped by function and by location
- **Variables**
  - Applying to groups or individual nodes
- **Roles**
  - Tie groups to templates and variables
  - Common or by function (edge routers, firewalls, etc.)
Variables

group_vars/

- all.yml
- ams.yml
- iad.yml
- edge.yml

host_vars/

- edge-01-ams.yml
- vpn-01-iad.yml
Templates

● Ansible uses Jinja2
  ○ Configuration text with embedded code (Python)
    ■ Conditionals, loops, etc.

● XML format
  ○ Because we had to
    ■ better support across versions of JunOS
  ○ But also allows for advanced checks
    ■ Easy to parse
    ■ Could use XSD schemas to validate
Template example

```
<host-name>{{ host_basename }}</host-name>
<domain-name>{{ domain_name }}</domain-name>
{% for domain in domain_search %}
<domain-search>{{ domain }}</domain-search>
{% endfor %}
{% if backup_router is defined %}
<backup-router>
  <address>{{ backup_router }}</address>
  <destination>0.0.0.0</destination>
</backup-router>
{% endif %}
<root-authentication>
  <encrypted-password>{{ root_password_hash }}</encrypted-password>
</root-authentication>
{% for name_server in name_servers %}
<name-server>
  <name>{{ name_server }}</name>
</name-server>
{% endfor %}
```
Access

- Enable NETCONF everywhere
  - Enable detailed logging
- Set up SSH public keys in every device
  - Read-only account (for collecting “facts”)
    - Allows non-admins to test their builds
  - Read-write account (for deploying)
  - Always encrypt private keys!
Operations: build

- Runs Ansible playbook that:
  - Gets “facts” from each device
    - Model, OS version, hardware info.
  - Renders each template using variables and facts
  - Combines multiple outputs into a single file
  - Validates XML
    - Basic parsing check at the moment
    - Plan to add more elaborate checks
  - Does every interface description match the naming convention?
Operations: test

- Take each configuration file and perform a dry run
  - aka commit-check in JunOS
  - Gather diffs from each device
    - or report errors!
  - Combine diffs to create a pretty Gist
  - Send Gist URL to net admins
Operations: deploy

- Runs Ansible playbook that:
  - Sends configs to all devices
    - using NETCONF in our case
  - If there are changes, commits those
    - If there are no changes, device is unaffected
  - Notifies NOC
  - Triggers audit
    - Run RANCID, for example
Customizations

- We modified the Ansible Junos module to:
  - Allow us to do dry-runs
    - `--check` in Ansible
    - `commit-check` in Junos
  - Specify an external file to save diffs
  - Changes incorporated into v1.1.0
Deploy enhancements

- Plan to use the Ansible API to add some smarts to the deploy operation
  - Handle changes to multiple data centers
  - Exploring the use of Rundeck to handle the deploy job, instead of running directly from Jenkins
    - More control, flexibility
Implementation Approach

- Start simple
  - Cover the most common parts first:
    - e.g. User accounts, NTP, DNS, SNMP, common prefix lists, etc.
  - Work towards 100% coverage incrementally
    - Slow process until everything is standardized
- The template becomes the policy
  - Perform periodic dry-runs and notify of any diffs
Challenges

● Cultural change
  ○ Requires us to think differently
  ○ Familiarity with source control, Ansible, etc.

● Dry-runs fail because someone has the lock
  ○ Nuisance until 100% templated

● Approval while network admins are not around?
  ○ Off hours emergencies, etc.
Limitations

- Platform APIs not standard (yet)
  - Will they ever be?
  - Much more challenging in multi-vendor environments
- Probably can’t do network-wide atomic changes
  - Network state is inter-dependent
  - Not good to leave in inconsistent state
To Consider

- Staging environment
  - Physical vs. Virtual
  - How to *really* test network configs?

- Explore
  - Inventory from database
  - Variables from database
  - REST API
Looking ahead

● Watch new tools
  ○ e.g. Schprokits (Jeremy Schulman)
    ■ Like Ansible, but more network-focused
    ■ Solves the problem of multiple APIs

● Things will likely change dramatically
  ○ Open platforms
    ■ Software not tied to hardware
  ○ Overlay vs. Underlay
    ■ Increasingly dumber network and more intelligent hypervisor, etc.
A note on network design

- Automation is great, but...
  - Can we avoid touching the network in the first place?
  - Is there tight coupling between the network and the servers/services?
  - Design with this in mind
Sharing

https://github.com/dyninc/kipper-demo
Thank you

cvicente@dyn.com