ANSIBLE

ansible.com
+1 800-825-0212

WHITEPAPER

Scaling and Performance of the Ansible Management Toolchain

INTRODUCTION

Ansible is an automation platform that fits a wide variety of use cases and purposes. This document exists to help users understand the parameters of how Ansible operates on larger infrastructures and in environments seeking to reduce the duration of IT operational tasks and reduce total overall human time involved. Whether you have 500 nodes, or many thousands, this document will help you best understand how to maximize Ansible (and IT automation in general) at your organization.

NOTE: This document does assume some understanding of the Ansible system, but should still be largely useful to new and prospective users.

SCALING VS PERFORMANCE – HOW ARE THEY RELATED?

"Scaling" typically means how many machines can exist at once. "Performance" typically refers to how fast something can be executed.

Ultimately, these two items are interlinked – nearly everyone is interested in how many machines their infrastructure can grow to and how fast they can update all of those machines at once. Similarly, one of the easiest things for an IT architecture to do when it is facing performance issues is to add more nodes, though in adding more nodes, management complexity can be increased. Scaling and performance of IT are basically the same issue.

This whitepaper discusses how to optimize your IT infrastructure to allow it to take best advantage of automation, and how to best choose your automation strategy to help you update your machines as fast as possible within the constraints of your network and operational requirements.

People and processes are vitally important in understanding this – think about your workflow, and what you are trying to optimize for.

In many cases, the ultimate goal of most people will be the ability to support infrastructure scale – to manage the capacity of machines that they need to have to support business objectives. How much you can scale with Ansible is basically infinite – if your IT systems are arranged in the right way and you choose the right approaches. You can start small, but managing scale from one laptop controlling all of your nodes is not optimal. Eventually you'll need to think about some higher-order issues.

Other users, however, will be more concerned with speed (performance) and the need to keep outage windows down to a minimum amount of time. Depending on use case, Ansible and built-in rolling updates can help you actually take it to zero – and we'll talk about ways to reduce the total time for the overall update. In cases of continuous deployment, the rolling update cycle time matters even less – having a shorter cycle time increases the amount of deploys you can do per day, but not everyone wants to deploy more than 4 times an hour. If you do, great – we'll also share ways to make your automation cycles occur even more quickly.

PACKAGE BANDWIDTH

Regardless of the management technology used, the #1 most important consideration should always be what servers beyond the machines being updated or managed are involved in the update or management process. There is going to be a limit to how many machines you can simultaneously update, and you will arrive at this as a matter of trial and error – and it's not an Ansible imposed limit. The first step to solving this is to make sure most of your data is already local to your network. Having all of your machines hit server.example.com to download the latest tarball of an application is abusive.

At the most basic level, assume an infrastructure of N nodes, running a stock OS distribution. In one scenario, they may hit the outside world for all package updates. In other scenario, they may use a local mirror. In the second scenario, not only will the install be significantly faster, but outside bandwidth will be greatly reduced. This includes such operations as yum or apt updates, downloading of tarballs, or accessing content on github.com. If you are repeating tasks, or have even a small number of servers that would download the same content twice, being respectful of the upstream servers and setting up local mirrors of content is the #1 thing you can do to speed up your deployments. Tools like "createrepo" in yum (to create a local mirror) or "apt-cacher-ng" (for apt, to create a dynamic cache) are good starting points.

As your infrastructure grows larger, you may find a single local mirror of external content is no longer sufficient. In this case, consider load balanced update mirrors or torrent based download systems for large files (squid, varnish, or even apt-cacher-ng as mentioned above). An example of a torrent setup might be using something like Murder, at https://github.com/lg/murder

Management infrastructures may also involve mirroring on a top-of-rack unit or control node, to create many small local mirrors rather than one large one, thus avoiding some of the need for the torrent approach.

USE CASES DEFINED

The needs of what to optimize in terms of management vary based on use case.

OS PROVISIONING

OS provisioning involves spinning up new cloud instances. In these cases, the number of orchestration commands run through Ansible are likely fixed and small, though a rapidly spinning up number of instances (say spawning 200 machines at one time) making requests on first boot can tax network infrastructure – and it's even worse if they are all doing kickstarted net-installs from distribution mirrors.

CONFIGURATION MANAGEMENT (CM)

Configuration management refers to the configuration of basic OS services and can often extend to systems services applications such as Apache, Memcache, or setting up a mail server (etc). In general configuration management tasks are heavy on using OS packaging commands and hitting network sources for installation packages, but are also often heavily parallelized (many servers at once). When applications deployed on these operating systems are packaged in native OS packages, the lines often blur with Application Deployment – but not always. In these scenarios you are probably more interested in maximizing parallelism than in other use cases.

APPLICATION DEPLOYMENT AND UPGRADES

Application deployment refers to deploying business applications on top of an already configured operating system. In addition to installing OS packages from configured mirrors, often installation will involve retrieving software from version control systems or build artifacts from systems like Jenkins. When updating Applications, there are usually two workflows. "Outage Windows" involve the entire application server going down for an amount of time the user would like to minimize, while a rolling update involves taking some of the servers down at a time, and utilizing load balancing and monitoring integrations to leave a web service fully operational throughout the procedure. While you may want a rolling update to execute quicker, timing is usually not as important if the service can remain externally accessible. The network demands on a rolling update (especially in terms of package and source control access) are much lower than a simultaneous update.

ORCHESTRATION

Orchestration is a general purpose catch-all, but could include arbitrary IT tasks like applying a hot fix or rebooting a server. It could also include a higher-level process that applies OS configuration management, reboots servers, waits for them to complete, and so on. It also involves the idea of controlling what servers update when, which is also a scenario that weighs heavily in Application Deployment and Upgrades of multi-tier architectures. Orchestration systems generally seek to minimize the total time of the overall update process and to maximize parallelism, but also usually execute less total overall steps.

OPERATIONAL WORKFLOWS

Different organizations use their management software in different ways (often many of the cases below, rather than just one). The characteristics required of the management solution vary based on use case, so it's important to understand the differences.

DEVELOPER ENVIRONMENTS

Generally speaking, in developer use cases, an engineer will request the provisioning of virtual machines and cloud resources in small numbers. In these scenarios both update/package bandwidth and management parallelism needs are very small. The developers may update the systems, but again, the total number of systems are small. In ideal cases, the workflow to update and deploy the development environment very closely tracks the production automation system.

PROVISIONING ONLY

In a provisioning only use case, a new project has started or a need for more capacity has been requested. Machines will be spun up to fulfill demand. In these cases, generally speaking, the number of instances added at any one time are often small. In cases where they are large, for instance, reinstallation of a Top 500 supercomputer cluster, the needs for update bandwidth often dictate that the update should be staged, perhaps reinstalling many dozens and or hundreds of machines at a time rather than the entire network. These concerns are independent of the management toolchain and the management toolchain is usually not stressed. Often provisioned nodes are set up in a base capacity for other teams to take "the rest of the way", which is often common in MSP or cloud environments.

NEEDS ORIENTED VS PERIODIC REMEDIATION

There are differing schools of thought as to whether machines should be configured when changes are required or whether the configuration system should always enforce a desired state. Both schools of thought are equally valid depending on application.

If following a "Needs Oriented" approach, a machine may be managed by the automation when rolling out a new service or when a known configuration change needs to be applied. This type of approach fits well with the "push-based" topology below, and generates low stress on the network, especially as parallelism can be easily controlled (see section on Parallelism) later on this document. The "Needs Oriented" approach often believes that automation repeatedly "fixing" a configuration (like a service that has crashed being restarted or a permission that is wrong) may hide other problems that need to be fixed, so ops staff may initiate change only when they know a change needs to be made.

If following a "Periodic Remediation" approach, the idea is the configuration system should run periodically (every 30 minutes is very common) to enforce that the current system matches configuration policy. The idea behind this scenario is that users (or employees not following the automation process) may change a system and the automation system should "put it back" to minimize drift. Note that the "needs oriented" approach can still fix drift but just doesn't run automatically. In a periodic remediation setup, this can be done in either a "Pull" mode (which may stress the network) and a "Push" mode that is centrally initiated and could utilize a rolling update. Both "Push" and "Pull" here are equally valid depending on characteristics of expected change and scale.

What approach is best for an organization depends on how likely the system is to be changed outside of the management software.

ARCHITECTURAL TOPOLOGIES

PUSH

Ansible's most natural form of managing remote machines is to push out descriptions of desired state in parallel to many servers at a time (see Parallelism in a later section). This approach allows control of how many updates are going on at one time, and as such, provides a great deal of control over network taxation and is the simplest to implement, and allows for easy integration with centralized reporting, such as the features offered in AWX.

Push topology is also very useful because it very tightly controls what information is disseminated to what nodes, and the results of a previous step on one node can influence the direction taken on other nodes. Push also generates a nice centralized report and can easily coordinate and share data between machines.

Push topology is also exceptional in doing multi-tier configuration, where as an example, configuration of a 3-tier application can be executed as quickly as possible, where as in a traditional pull-based architecture, the update cycle may occur in 90 minutes (30+30+30) as each tier must complete in turn, leading to costly outage links.

In short, push based modes allow for the most rapid configuration of multi-tier configuration, and are critical in enabling rolling updates and other processes where tight order-based operations between related groups of servers are required.

LOCAL

Local mode involves transporting Ansible playbooks to the box via some outside means (tarball, torrent, etc) and running Ansible without a server entirely. This mode allows theoretically infinite scale exempting update bandwidth, but does not offer any centralized reporting – and some effort is required to decide how to deploy it. Pull mode is an extension of local mode, so pure local is not commonly used. However, if you want to run configuration in a stand-alone environment (configure a server without networking drop-shipped to a tropical desert island), local mode is there for your IT enjoyment and is blazingly fast.

PULL (SCM BACKED)

Pull based modes rely on a crontab to execute a git (or other SCM) pull of configuration/automation content at periodic intervals. The ansible-pull utility included with ansible does exactly this. Once the configuration data is retrieved, ansible runs locally. This hybrid approach is well battle-tested for centrally managed massive-scale-out topologies on some of the web's largest properties (with Ansible and also other systems). Logs about the run must be harvested by some means, usually a pull-based Ansible retrieval. In the near future, pull based modes may be able to log to an AWX server. While applicability varies based on use case, in general, ansible-pull is almost always unnecessary in topologies of less than 500 nodes, and often unnecessary in much larger topologies (x<2000 nodes or more) depending on use case.

The concept of a lot of machines checking in at the same time is called a "thundering herd" scenario and is best handled by random delays between check-ins and a very robust system for managing package bandwidth. For this reason, the Callback system (below) or the Push-Pull Hybrid system can often be superior in cases where capacity is not sufficient to update the entire network at once. Ansible minimizes the thundering herd problem by just using the SCM as the remote source rather than a CPU-intensive server compile process, so more nodes can be serviced simultaneously.

PUSH-PULL HYBRID

A push-pull topology involves a regular ansible playbook invoking ansible-pull (or any other configuration method) rather than ansible-pull being invoked in cron. This is a bit of an unusual setup, but has the advantage of a minimal number of ansible tasks being executed over the network.

An example of this is (as of time of writing) Rackspace.com's Ansible setup, where Ansible prepares a local environment, orchestration initiates the download of content, runs configuration steps, and then finally runs some remaining post-configuration steps via Ansible. In this setup, full configuration of a very large infrastructure can be executed in a matter of minutes.

CALLBACK-DRIVEN

A callback driven approach is appropriate for auto-scaling use cases, where cloud nodes from images could spin up at any time and need to request to be configured using the "most current configuration", without the configuration of the cloud nodes being needed to be baked into the image.

In this approach, Ansible AWX is used, and a first boot script is added to the base image to request the "callback job trigger" URL provided by AWX. Once requested via curl or wget, the system that has phoned home is pushed-to by Ansible and fully configured. The size of the job queue in AWX is tunable, to control how many systems are being configured at once. Because of this queue, this setup allows for periodic remediation of architectures of very large size, without resulting in "thundering herd" update scenarios.

This approach may also be appropriate for cloud nodes for other reasons – such as systems that are spun down and restarted when required, and when they spin up they need to get the latest configuration after being not configured by IT automation for many months.

PACKAGE BANDWIDTH AND SAAS CONTENTION

As alluded to above, the most important concern in scaling any automation system is making sure there is enough bandwidth and network capacity for all systems to update (or grab their required content at once).

Like package bandwidth, another chokepoint may be the need to talk to web services on the network. For instance, if updating 1000 systems at once, and they all have to talk to a REST API at the same time, it is very easy for them to overwhelm the REST API. In scenarios like these, controlling parallelism and encouraging rolling updates (or using the callback system above to moderate simultaneous requests) is a good idea.

Setting the "serial" parameter in the playbook to a smaller number when talking to these resource constrained services is a good idea.

ROLLING UPDATES

Rolling updates (as alluded to previously) are the idea that N out of M nodes are updated at a given time, talking to load balancers and monitoring prior to the update to remove them from a pool, and putting them back in pools prior to testing and restarting services.

Rolling update topologies are not only great at eliminating user downtime on accessing a web service, but they are also a great way of updating a finite capacity of machines at a given time. Time-to-completion concerns are also lessened as there are no site outages. (Usually the longest part of any update cycle will be in package updates and service restarts).

Rolling updates are also great because they bring scale with them. As they minimize bandwidth and parallelism concerns of various services on your network (including Ansible) they allow you addressability into very large scale even with constrainted external resources.

Using the “serial: N” keyword in Ansible also has other advantages, ensuring that all of your nodes are not halfway through a configuration and hit a problem allows you to use the first part of a serial batch as a “canary” to test your update as it goes out into the wild. (Of course, you should also have a stage environment!)

PACKAGE TRANSACTION OPTIMIZATIONS

Ansible is very smart when it comes to package installations – it tries to minimize the total number of package transactions used, recognizing that dependency solving, fetching the latest metadata, and so on, can be slow operations.

With yum and apt (and some others), when a “with_items” statement in Ansible is used, all packages listed in that list of items are batched together in a single yum/apt transaction.

This allows update times in Ansible to be significantly shorter than many other automation systems.

When using apt in particular, there are also parameters available to decide how often to update the apt-cache. This can be set so that repeated playbook runs might not need to update the cache unneccessarily.

FACT GATHERING OPTIMIZATIONS

In plays that do not need to make use of any Ansible facts, or where facts have already been gathered for the hosts in a previous play, fact gathering can be turned off on the play to save a few seconds.

Set “gather_facts: False” in the header of a play to achieve this effect.

ASYNC FOR LONG RUNNING JOBS

For long running operations in Ansible, such as those that may exceed an SSH timeout, using async features (see documentation) allows operations to be run in the background and optionally polled until completion every selected number of seconds.

Usage of async without a poll interval is called “fire and forget” mode, which dispatches a background task that nothing needs to wait on.

Consider async mode (likely with polling) for steps like online firmware upgrades or long-running system-wide packages updates.

NODE PARALLELISM

Ansible uses the “–forks” parameter in push mode to decide how many machines to talk to at once.

In general, you can raise this value to reasonably high levels on well-resourced hardware and virtual machines.

Values as high as 750 can yield very reasonable results (talking to 750 hosts at a time).

Setting this value lower is advised on poorly speced virtual machines or if excessive resource usage is noticed.

MANAGING SERVICE RESTARTS

Automation is faster when you restart services less. Rather than inserting manual "service: name=foo state=restarted" steps, notify handlers to restart services only when required files change. This also means less service interruptions on the managed machines.

UNDERSTANDING ANSIBLE CONNECTION TYPES

The different network connection types in Ansible behave differently and will result in executing jobs at slightly different rates.

LOCAL

Local mode affects the local system only. It is exceedingly quick and is the basis behind "pull" and "push-pull" topologies above. Using local mode is the quickest way to get a series of tasks completed, but is usually unnecessary for most users.

PARAMIKO

Paramiko is a Python-implementation of SSH. As of Ansible 1.2.X, it is only used by default on Enterprise Linux 6 or earlier systems that do not have "Control Persist" capability in SSH. Paramiko achieves what we'll call "baseline" performance, but is less optimal because it will connect seperately for each task and will not reuse connections.

SSH WITHOUT CONTROL PERSIST

SSH without Control Persist is used when on a Enterprise Linux 6 or earlier system and the SSH connection type is explicitly selected. It has performance characteristics similar to Paramiko (often sometimes slightly slower) but also has the advantage of being reasonably tunable.

SSH WITH CONTROL PERSIST

SSH with Control Persist is the default for most Ansible users (Fedora, Debian, Ubuntu, OS X, etc) where the version of OpenSSH supplied is new enough. Control Persist keeps SSH connections active between operations for a configurable timeout, which can be set in your SSH configuration file or in the Ansible configuration file. For best results, use a long timeout, such as 30 minutes or an hour. SSH with Control Persist is roughly 2.5 faster than without (or relative to Paramiko) in average network conditions.

ACCELERATED MODE

Ansible contains a connection type called "accelerated" which uses a encrypted (AES) socket using an temporary shared secret that is generated every 30 minutes. Ansible uses keyczar for encryption rather than implementing any of it's own encryption in this case, and uses SSH for key exchange. This sets up a temporary daemon that runs for a specified amount of time (default 30 minutes) and allows connections only from the user that initiated the temporary daemon. Starting another daemon will terminate the previous daemon. Accelerated mode has some minimal dependencies on the remote nodes (which Ansible does not) – namely the encryption libraries – and achieves an additional 2.5x speedup over SSH with ControlPersist under average network conditions, and 10x speedups over Paramiko.

OPERATIONAL SCALING

OPERATIONAL WORKLOAD PERFORMANCE

While how many machines can be effectively managed by the management solution is what we've concentrated on so far, one of the more important considerations that Ansible focuses on is how resources are consumed on remote nodes. Because Ansible is an agentless solution, when Ansible is not actively managing a system, nothing is running on the remote system. This means there are no extra CPU cycles consumed by Ansible, nor any concerns about daemon memory utilization.

Historically speaking, one open source solution we encountered could consume 400MB of memory per node for running the management agent – which expanded over 20 virtual machines per node (for instance) approaches 8GB of memory consumption per physical device. This number of Ansible is zero, and because it is agentless, it is immune to any possible memory leaks. As a result the performance of the managed systems themselves is increased.

MANAGING THE MANAGEMENT

As the number of nodes under management increase, with most solutions, so do the complexities of keeping the management software running. In systems that rely on management agents, on a large infrastructure it's quite possible many of the agents are down (not running or crashed) and you need some out of band way to restart them. Or possibly they are running at different versions. Ansible avoids this problem by not requiring any remote agents (or even software) on remote machines, as the modules it transfers are self sufficient. As a result, you can always manage all of your machines without worrying about connectivity of the management channel. There are also no open ports the machines under management must sustain open to the management server.

This is especially nice in upgrade scenarios – moving to a new Ansible version requires only updating the "control machine" with no changes required at all on the remote nodes – there are no agents to update.

MANAGEMENT PROXIMITY AND MULTI-DATACENTER

When using remote management techniques, just as it is important to maintain local copies of installation content, the management system also can update things faster if a control node is located inside the network.

For example, if managing a fleet of machines on EC2, run Ansible inside EC2, and preferably within the EC2 zone for maximum benefits. This will also save you bandwidth.

CONCLUSIONS

How you decide to manage your network of computers will vary dramatically based on what sort of operations you are wishing to conduct, network size, geographic distribution, and whether you wish to engage in an outage period or implement rolling updates.

Regardless of structure, the most important consideration is going to be network bandwidth for package distribution. In controlling bandwidth considerations, workflows such as rolling updates or AWX callbacks may be greatly advantageous.

As an easy way to minimize update time regardless of operational choices, consider SSH with Control Persist or accelerated mode for quick update life cycles.

Regardless of the above characteristics, Ansible doesn't punish you when you scale by giving you more management to manage – having no agents to keep running and just relying on OpenSSH – which you already have installed simplifies the operational lifecycle of the management lifecycle itself, making it a largely transparent part of your IT process.

If you would like to discuss infrastructure strategies further, you can contact us at **info@ansible.com**

AUTHORS AND REVIEWERS

Michael DeHaan, CTO, AnsibleWorks
James Cammarata, Senior Software Engineer, AnsibleWorks
Jesse Keating, Linux Systems Engineer, Rackspace