Configuration management and automation tools like SaltStack are great, they allow us to deploy a configuration change to thousands of servers with out much effort. However, while these tools are powerful and give us greater control of our environment they can also be dangerous. Since you can roll out a configuration change to all of your servers at once, it is easy for that change to break all of your servers at once.

In today's article I am going to show a few ways you can run a SaltStack "highstate" across your environment, and how you can make those highstate changes a little safer by staggering when servers get updated.

**Why stagger highstate runs**

Let's imagine for a second that we are running a cluster of 100 webservers. For this webserver cluster we are using the following nginx state file to maintain our configuration, ensure that the nginx package is installed and the service is running.

nginx:

pkg:

- installed

service:

- running

- watch:

- pkg: nginx

- file: /etc/nginx/nginx.conf

- file: /etc/nginx/conf.d

- file: /etc/nginx/globals

/etc/nginx/globals:

file.recurse:

- source: salt://nginx/config/etc/nginx/globals

- user: root

- group: root

- file\_mode: 644

- dir\_mode: 755

- include\_empty: True

- makedirs: True

/etc/nginx/nginx.conf:

file.managed:

- source: salt://nginx/config/etc/nginx/nginx.conf

- user: root

- group: root

- mode: 640

/etc/nginx/conf.d:

file.recurse:

- source: salt://nginx/config/etc/nginx/conf.d

- user: root

- group: root

- file\_mode: 644

- dir\_mode: 755

- include\_empty: True

- makedirs: True

Now let's say you need to deploy a change to the nginx.conf configuration file. Making the change is pretty straight forward, we can simply change the source file on the master server and use salt to deploy it. Since we listed the nginx.conffile as a watched state, SaltStack will also restart the nginx service for us after changing the config file.

To deploy this change to all of our servers we can run a highstate from the master that targets every server.

# salt '\*' state.highstate

One of SaltStack's strengths is the fact that it performs tasks in parallel across many minions. While that is a useful feature for performance, it can be a bit of problem when running a highstate that restarts services across all of your minions.

The above command will deploy the configuration file to each server and restart nginx on all servers. Effectively bringing down nginx on all hosts at the same time, even if it is for just a second that restart is probably going to be noticed by your end users.

To avoid situations that bring down a service across all of our hosts at the same time, we can stagger when hosts are updated.

**Staggering highstates**

**Ad-hoc highstates from the master**

Initiating highstates is usually either performed ad-hoc or via a scheduled task. There are two ways to initiate an ad-hoc highstate, either via the salt-callcommand on the minion or the salt command on the master. Running the salt-call command on each minion manually naturally avoids the possibility of restarting services on all minions at the same time as it only affects the minion where it is run from. The salt command on the master however can if given the proper targets be told to update all hosts, or only a subset of hosts at a given time.

The most common method of calling a highstate is the following command.

# salt '\*' state.highstate

Since the above command runs the highstate on all hosts in parallel this will not work for staggering the update. The below examples will cover how to use the salt command in conjunction with SaltStack features and minion organization practices that allow us to stagger highstate changes.

**Batch Mode**

When initiating a highstate from the master you can utilize a feature known as batch mode. The --batch-size flag allows you to specify how many minions to run against in parallel. For example, if we have 10 hosts and we want to run a highstate on all 10 but only 5 at a time. We can use the command below.

# salt --batch-size 5 '\*' state.highstate

The batch size can also be specified with the -b flag. We could perform the same task with the next command.

# salt -b 5 '\*' state.highstate

The above commands will tell salt to pick 5 hosts, run a highstate across those hosts and wait for them to finish before performing the same task on the next 5 hosts until it has run a highstate across all hosts connected to the master.

**Specifying a percentage in batch mode**

Batch size can take either a number or a percentage. Given the same scenario, if we have 10 hosts and we want to run a highstate on 5 at a time. Rather than giving the batch size of 5 we can give a batch size of 50%.

# salt -b 50% '\*' state.highstate

**Using unique identifiers like grains, nodegroups, pillars and hostnames**

Batch mode picks which hosts to update at random, you may yourself wanting to upgrade a specific set of minions first. Within SaltStack there are several options for identifying a specific minion, with some pre-planning on the organization of our minions we can use these identifiers to target specific hosts and control when/how they get updated.

**Hostname Conventions**

The most basic way to target a server in SaltStack is via the hostname. Choosing a good hostname naming convention is important in general but when you tie in configuration management tools like SaltStack it helps out even more ([see this blog post for an example](https://blog.serverdensity.com/picking-server-hostnames/)).

Let's give another example where we have 100 hosts, and we want to split our hosts into 4 groups; group1, group2, group3 and group4. Our hostname will follow the convention of webhost<hostnum>.<group>.example.com so the first host in group 1 would be webhost01.group1.example.com.

Now that we have a good naming convention if we want to roll-out our nginx configuration change and restart to these groups one by one we can do so with the following salt command.

# salt 'webhost\*group1\*' state.highstate

This command will only run a highstate against hosts that have a hostname that matches the 'webhost\*group1\*' pattern. Which means that only group1's hosts are going to be updated with this run of salt.

**Nodegroups**

Sometimes you may find yourself in a situation where you cannot use the hostname to identify classes of minions and the hostnames can't easily be changed, for whatever reasons. If descriptive hostnames are not an option than one alternate solution for this is to use nodegroups. Nodegroups are an internal grouping system within SaltStack that will let you target groups of minions by a specified name.

In the example below we are going to create 2 nodegroups for a cluster of 6 webservers.

**Defining a nodegroup**

On the master server we will define 2 nodegroups, group1 and group2. To add these definitions we will need to change the /etc/salt/master configuration file on the master server.

# vi /etc/salt/master

**Find:**

##### Node Groups #####

##########################################

# Node groups allow for logical groupings of minion nodes.

# A group consists of a group name and a compound target.

#

# group1: 'L@foo.domain.com,bar.domain.com,baz.domain.com and bl\*.domain.com'

# group2: 'G@os:Debian and foo.domain.com'

**Modify To:**

##### Node Groups #####

##########################################

# Node groups allow for logical groupings of minion nodes.

# A group consists of a group name and a compound target.

#

group1: 'L@webhost01.example.com,webhost02.example.com and webhost03.example.com'

group2: 'L@webhost04.example.com,webhost05.example.com and webhost06.example.com'

After modifying the /etc/salt/master we will need to restart the salt-masterservice

# /etc/init.d/salt-master restart

**Targeting hosts with nodegroups**

With our nodegroups defined we can now target our groups of minions by passing the -N <groupname> arguments to the salt command.

# salt -N group1 state.highstate

The above command will only run the highstate on minions within the group1 nodegroup.

**Grains**

Defining unique grains is another way of grouping minions. Grains are kind of like static variables for minions in SaltStack; by default grains will contain information such as network configuration, hostnames, device information and OS version. They are set on the minions during start time and they do not change, this makes them a great candidate to use to identify groups of minions.

To use grains to segregate hosts we must first create a grain that will have different values for each group of hosts. To do this we will create a grain called group the value of this grain will be either group1 or group2. If we have 10 hosts, 5 of those hosts will be given a value of group1 and the other 5 will be given a value of group2.

There are a couple of ways to [set grains](http://docs.saltstack.com/en/latest/topics/targeting/grains.html#grains-in-the-minion-config), we can do it either by editing the /etc/salt/minion configuration file or the /etc/salt/grains file on the minion servers. I personally like putting grains into the /etc/salt/grains file and that's what I will be showing in this example.

**Setting grains**

To set our group grain we will edit the /etc/salt/grains file.

# vi /etc/salt/grains

**Append:**

group: group1

Since grains are only set during start of the minion service we will need to restart the salt-minion service.

# /etc/init.d/salt-minion restart

**Targeting hosts with grains**

Now that our grain is set we can target our groups using the -G flag of the saltcommand.

# salt -G group:group2 state.highstate

The above command will only run the highstate function on minions where the grain group is set to group2

**Using batch-size and unique identifiers together**

At some point, after creating nodegroups and grouping grains you may find that you still want to deploy changes to only a percentage of those minions.

Luckily we can use --batch-size and nodegroup and grain targeting together. Let's say you have 100 webservers, and you split your webservers across 4 nodegroups. If you spread out the hosts evenly each nodegroup would have 25 hosts within it, but this time restarting all 25 hosts is not what you want. Rather you would prefer to only restart 5 hosts at a time, you can do this with batch size and nodegroups.

The command for our example above would look like the following.

# salt -b 5 -N group1 state.highstate

This command will update the group1 nodegroup, 5 minions at a time.

**Scheduling updates**

The above examples are great for ad-hoc highstates across your minion population, however that only fixes highstates being pushed manually. By scheduling highstate runs, we can make sure that hosts get the proper configuration automatically without any human interaction, but again we have to be careful with how we schedule these updates. If we simple told each minion to update every 5 minutes, those updates would surely overlap at some point.

**Using Scheduler to schedule updates**

The SaltStack scheduler system is a great tool for scheduling salt tasks; especially the highstate function. You can configure scheduler in SaltStack two ways, by appending the configuration to the /etc/salt/minion configuration file on each minion or by setting the schedule configuration as a pillar for each minion.

Setting the configuration as a pillar is by far the easiest, however the version of SaltStack I am using 0.16 has a bug where setting the scheduler configuration in the pillar does not work. So the example I am going to show is the first method. We will be appending the configuration to the /etc/salt/minionconfiguration file, we are also going to use SaltStack to deploy this file as we might as well tell SaltStack how to manage itself.

**Creating the state file**

Before adding the schedule we will need to create the state file to manage the minion config file.

**Create a saltminion directory**

We will first create a directory called saltminion in /srv/salt which is the default directory for salt states.

# mkdir -p /srv/salt/saltminion

**Create the SLS**

After creating the saltminion directory we can create the state file for managing the /etc/salt/minion configuration file. By naming the file init.slswe can reference this state as saltminion in the top.sls file.

# vi /srv/salt/saltminion/init.sls

**Insert:**

salt-minion:

service:

- running

- enable: True

- watch:

- file: /etc/salt/minion

/etc/salt/minion:

file.managed:

- source: salt://saltminion/minion

- user: root

- group: root

- mode: 640

- template: jinja

- context:

saltmaster: master.example.com

{% if "group1" in grains['group'] %}

timer: 20

{% else %}

timer: 15

{% endif %}

The above state file might look a bit daunting but it is pretty easy, the first section ensures that the salt-minion service is running and enabled. It also watched the /etc/salt/minion config file and if it changes than salt will restart the service. The second section is where things get a bit more complicated. The second section manages the /etc/salt/minion configuration file, most of this is standard salt stack configuration management. However, you may have noticed a part that looks a bit different.

{% if "group1" in grains['group'] %}

timer: 20

{% else %}

timer: 15

{% endif %}

The above is an example of using jinja inside of a state file. You can use the jinja templating in SaltStack to create complicated statements. The above will check if the grain "group" is set to group1, if it is set then it will add set the timer context to 20. If it is not set than it will default to a context of 15.

**Create a template minion file**

In the above salt state we told SaltStack that the salt://saltminion/minion file is a template, and that template file is a jinja template. This tells SaltStack to read the minion file and use the jinja templating language to parse it. The items under context are variables being passed to jinja while processing the file.

At this point it would probably be a good idea to actually create the template file, to do this we will start with a copy from the master server.

# cp /etc/salt/minion /srv/salt/saltminion/

Once we copy the file into the saltminion directory we will need to add the appropriate jinja markup.

# vi /srv/salt/saltminion/minion

First we will add the saltmaster variable, which will be used to tell the minions which master to connect to. In our case this will be replaced with master.example.com.

**Find:**

#master: salt

**Replace with:**

master: {{ saltmaster }}

After adding the master configuration, we can add the scheduler configuration to the same file. We will add the following to the bottom of the minionconfiguration file.

**Append:**

schedule:

highstate:

function: state.highstate

minutes: {{ timer }}

In the scheduler configuration the timer variable will be replaced with either 15 or 20 depending on the group grain that is set on the minion. This will tell the minion to run a highstate every 15 or 20 minute, that should give approximately 5 minutes between groups. The timing of this may need adjustment depending on the environment. When dealing with large amounts of servers you may need to build in a larger time between highstates between the groups.

**Deploying the minion config**

Now that we have created the minion template file, we will need to deploy it to all of the minions. Since they don't already automatically update we can run an ad-hoc highstate from the master. Because we are restarting the minion service we may want to use --batch-size to stagger the updates.

# salt -b 10% '\*' state.highstate

The above command will update all minions but only 10% of them at a time.

**Using cron on the minions to schedule updates**

An alternative to using SaltStacks scheduler is cron, the cron service was the default answer for scheduling highstates before the scheduler system was added into SaltStack. Since we are deploying a configuration to the minions to manage highstates, we can use salt to automate and managed this.

**Creating the state file**

Like with the scheduler option we will create a saltminion directory within the /srv/salt directory.

# mkdir -p /srv/salt/saltminion

**Create the SLS file**

There are a few ways you can create crontabs in salt, but I personally like just putting a file in /etc/cron.d as it makes the management of the crontab as simple as managing any other file in salt. The below SLS file will deploy a templated file /etc/cron.d/salt-highstate to all of the minions.

# vi /srv/salt/saltminion/init.sls

**Insert:**

/etc/cron.d/salt-highstate:

file.managed:

- source: salt://saltminion/salt-highstate

- user: root

- group: root

- mode: 640

- template: jinja

- context:

updategroup: {{ grains['group'] }}

**Create the cron template**

Again we are using template files and jinja to determine which crontab entry should be used. We are however performing this a little differently. Rather than putting the logic into the state file, we are putting the logic in the source file salt://saltminion/salt-highstate and simply passing the grains['group']value to the template file in the state configuration.

# vi /srv/salt/saltminion/salt-highstate

**Insert:**

{{ if "group1" in updategroup }}

\*/20 \* \* \* \* root /usr/bin/salt-call state.highstate

{{ else }}

\*/15 \* \* \* \* root /usr/bin/salt-call state.highstate

{{ endif }}

One advantage of cron over salt's scheduler is that you have a bit more control of when the highstate runs. The scheduler system runs over an interval with the ability to define seconds, minutes, hours or days. Whereas cron gives you that same ability but also allows you to define complex schedules like, "only run every Sunday if it is the 15th day of the month". While that may be a bit overkill for most, some may find that the flexibility of cron makes it easier to avoid both groups updating at the same time.

**Using cron on the master to schedule updates with batches**

If you want to run your highstates more frequently and avoid conditions where everything gets updated at the same time. Rather than scheduling updates from the minions, one could schedule the update from the salt master. By using cronon the master, we can use the same ad-hoc salt commands as above but call them on a scheduled basis. This solution is somewhat a best of both worlds scenario. It gives you an easy way of automatically updating your hosts in different batches and it allows you to roll the update to those groups a little at a time.

To do this we can create a simple job in cron, for consistency I am going to use /etc/cron.d but this could be done via the crontab command as well.

# vi /etc/cron.d/salt-highstate

**Insert:**

0 \* \* \* \* root /usr/bin/salt -b 10% -G group:group1 state.highstate

30 \* \* \* \* root /usr/bin/salt -b 10% -G group:group2 state.highstate

The above will run the salt command for group1 at the top of the hour every hour and the salt command for group2 at the 30th minute of every hour. Both of these commands are using a batch size of 10% which will tell salt to only update 10% of the hosts in that group at a time. While this method might have some hosts in group1 being updated while group2 is getting started, overall it is fairly safe as it ensures that the highstate is only running on at most 20% of the infrastructure at a time.

One thing I advise it to make sure that you also segregate these highstates by server role as well. If you have a cluster of 10 webservers and only 2 database servers, all of those servers are split amongst group1 and group2; with the right timing both databases could be selected for a highstate at the same time. To avoid this you could either have your "group" grains be specific to the server roles or setup nodegroups that are specific to server roles.

An example of this would look like the following.

0 \* \* \* \* root /usr/bin/salt -b 10% -N webservers1 state.highstate

15 \* \* \* \* root /usr/bin/salt -b 10% -N webservers2 state.highstate

30 \* \* \* \* root /usr/bin/salt -b 10% -N alldbservers state.highstate