

Introducing SaltStack



SALTSTACK

What is Salt?

Salt is a different approach to infrastructure management, founded on the idea that high-speed communication with large numbers of systems can open up new capabilities. This approach makes Salt a powerful multitasking system that can solve many specific problems in an infrastructure.

The backbone of Salt is the remote execution engine, which creates a high-speed, secure and bi-directional communication net for groups of systems. On **top** of this communication system, Salt provides an extremely fast, flexible and easy-to-use configuration management system called **Salt States** (like Puppet Resources).

Some Customers and Awards



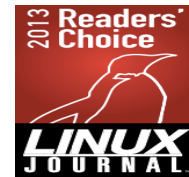
InfoWorld



GitHub



to
install



- A 'Minion' is the local agent.
- Master connects to a Minion by [ZeroMQ](#), or [SSH](#), to create a:
 - fast
 - persistent
 - PKI based
 - reliable network agents
- Master/Minion messages are coded as [MessagePack](#) format (like zipped JSON).
- Master runs a [ZeroMQ](#) file server to push configuration files and Python code (import) on the Minions.
- SaltStack is written in Python 2.6, mainly for Linux but it also supports Solaris and Windows.

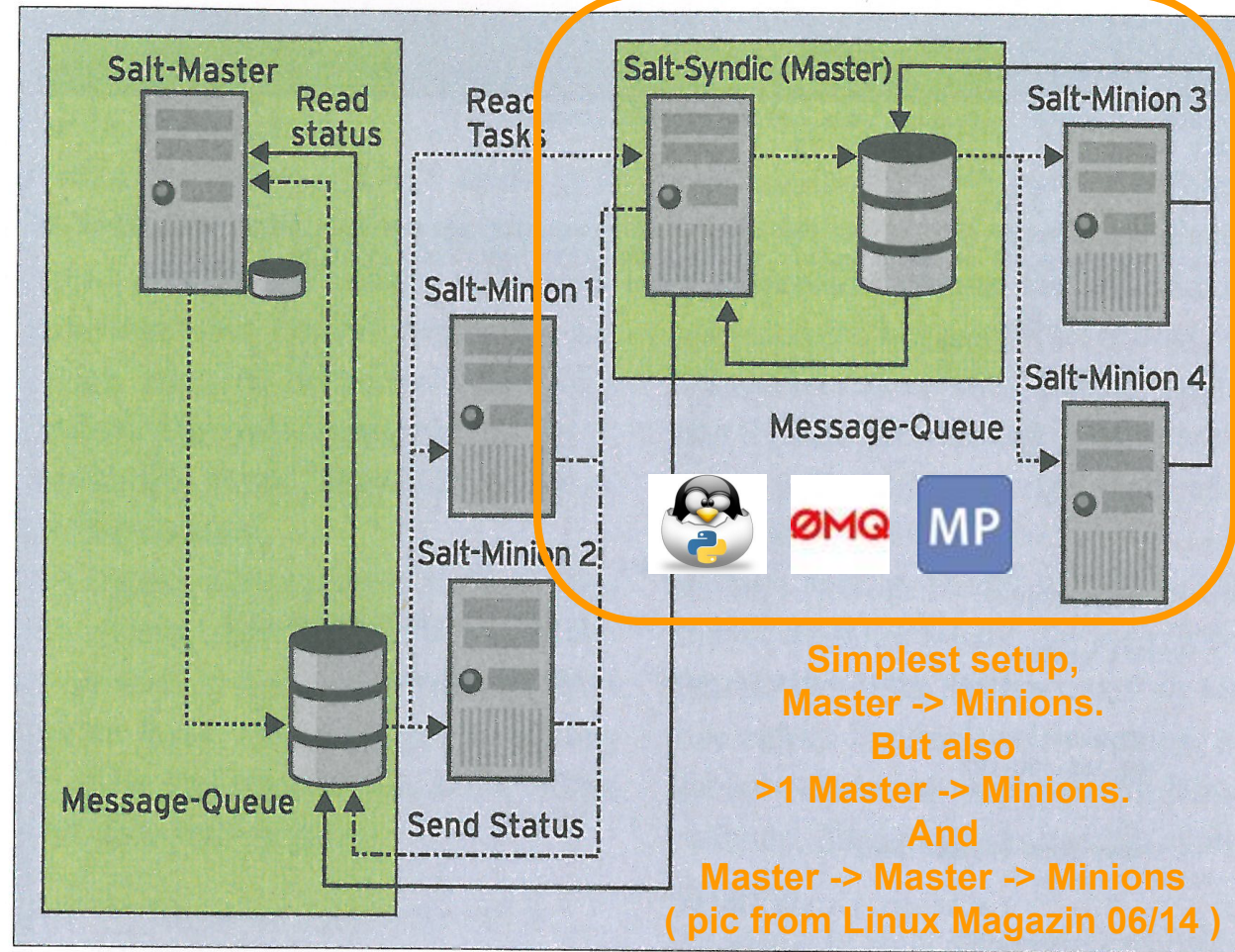


Abbildung 1: Die Salt-Architektur inklusive des häufig eingesetzten Zwischen-Masters Salt-Syndic im Überblick. Die durchgezogenen Linien kennzeichnen die Rückmeldungen, die Zielsysteme an die Master liefern.



- Each Minion publishes properties to the Master and to the other Minions e.g.:
 - *os*
 - *osrelease*
 - *kernelrelease*
 - *cpu_flags*, ...
 - called [Grains](#) (like Puppet facts).
- The Master dynamically selects the Minions that own certain Grains values to:
 - run an arbitrary Linux command (like **cexec/pdsh**)
 - run the predefined SaltStack [Execution Modules](#) (*disk.usage*, ..)
 - run distributed [Jobs](#) (like **nohup cexec/pdsh &**)
 - change the Minion state by applying the states files reported in the *top.sls* file.
 - either in the run or in the change case, the output can be plaintext or **JSON**.
- Minions states files can be written in:
 - **YAML** (ugly, but straightforward to write/read)
 - [Python](#) (my choice)
 - both use the predefined [Salt States](#) , i.e. the usual **File/Pkg/Service/User/Group/Mount** ... operations (like Puppet Resources)
- There is a handy Python API in front of the Master and the Minion.

```

root@t3vmui01 ~]# salt-call grains.items
local:
  MYGRAIN: ISMYGRAIN
  biosreleasedate: 07/09/2012
  biosversion: 6.00
  cpu_flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss syscall nx rdtscp lm constant_tsc up ida nonstop_tsc arat pni ssse3 cx16 sse4_1 sse4_2 popcnt lahfm_lahf_lm
  cpu_model: Intel(R) Xeon(R) CPU E5-2640 0 @ 2.50GHz
  cpusarch: x86_64
  defaultencoding: UTF8
  defaultlanguage: en_US
  domain: psi.ch
  external_ip: 192.33.123.209
  fqdn: t3vmui01.psi.ch
  fqdn_ip4:
    192.33.123.209
  fqdn_ip6:
  gpus:
    ('model': 'SVGA II Adapter', 'vendor': 'unknown')
  host: t3vmui01
  hwaddr_interfaces: ('sit0': '0.0.0.0', 'lo': '00:00:00:00:00:00', 'eth0': '00:50:56:95:00:25')
  id: t3vmui01
  ip_interfaces: ('sit0': [], 'lo': ['127.0.0.1'], 'eth0': ['192.33.123.209'])
  ipv4:
    127.0.0.1
    192.33.123.209
  ipv6:
    ::1
    fe80::250:56ff:fe95:25
  kernel: Linux
  kernelrelease: 2.6.18-371.6.1.el5
  localhost: t3vmui01
  master:
    t3admin01.psi.ch
    t3nagios.psi.ch
  mem_total: 1002
  nodename: t3vmui01
  num_cpus: 1
  num_gpus: 1
  os: ScientificLinux
  os_family: RedHat
  osarch: x86_64
  oscodename: Boron
  osfinger: Scientific Linux SL-5
  osfullname: Scientific Linux SL
  osmajorrelease:
    5
    7
  osrelease: 5.7
  path: /usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/root/bin
  productname: VMware Virtual Platform
  ps: ps -efH
  pythonpath:
    /usr/bin
    /usr/lib64/python2.6.zip
    /usr/lib64/python2.6
    /usr/lib64/python2.6/plat-linux2
    /usr/lib64/python2.6/lib-tk
    /usr/lib64/python2.6/lib-old
    /usr/lib64/python2.6/lib-dynload
    /usr/lib64/python2.6/site-packages
    /usr/lib/python2.6/site-packages
    /usr/lib/python2.6/site-packages/setuptools-0.6c11-py2.6.egg-info
  pythonversion: 2.6.8.final.0
  saltpath: /usr/lib/python2.6/site-packages/salt
  saltversion: 2014.1.1
  saltversioninfo: (2014, 1, 1)
  serialnumber: VMware-42 15 5d 99 29 4e d8 f4-63 e2 49 80 b0 11 8c 42
  server_id: 10040755
  shell: /bin/bash
  virtual: VMware
  vmware_version: 5.7.0
[root@t3vmui01 ~]# cat /etc/salt/grains
MYGRAIN: ISMYGRAIN

```

Minions Grains (like Puppet facts).

‘MYGRAIN’ was defined by me, at runtime.

Note the multi master definition.

In the next slide we’re using the Grain *nodename* to run the Execution Module *disk.usage* on a Minion.
i.e. we’ll run the *usage* function of */usr/lib/python2.6/site-packages/salt/modules/disk.py*

```
root@t3admin01 ~]# salt -G 'nodename:t3vmui01*' disk.usage
```

```
/:
-----
1K-blocks:
 9592544
available:
 5360768
capacity:
 41%
filesystem:
 /dev/md1
used:
 3720136
/boot:
-----
1K-blocks:
 988024
available:
 912648
capacity:
 3%
filesystem:
 /dev/md0
used:
 24376
/dev/ahm:
-----
1K-blocks:
 513032
available:
 513032
capacity:
 0%
filesystem:
 tmpfs
used:
 0
/opt:
-----
1K-blocks:
 1912144
available:
 1618024
capacity:
 11%
filesystem:
 /dev/md5
used:
 192132
/scratch:
-----
1K-blocks:
 22479104
available:
```

docs.saltstack.com/en/latest/ref/modules/all/salt.modules.disk.html#module-salt.modules.disk

21.16.34. salt.modules.disk

Module for gathering disk information

salt.modules.disk.blkid(device=None)

Return block device attributes: UUID, LABEL, etc.

CLI Example:

```
salt '*' disk.blkid
salt '*' disk.blkid /dev/sda
```

salt.modules.disk.inodeusage(args=None)

Return inode usage information for volumes mounted on this minion

CLI Example:

```
salt '*' disk.inodeusage
```

salt.modules.disk.percent(args=None)

Return partition information for volumes mounted on this minion

CLI Example:

```
salt '*' disk.percent /var
```

salt.modules.disk.usage(args=None)

Return usage information for volumes mounted on this minion

CLI Example:

```
salt '*' disk.usage
```

Execution Module example:

master ~# salt -G 'nodename:t3vmui01*' disk.usage

Another 'disk.usage' remote execution on that Minion, this time by IPython + the SaltStack Python API

```
master # /usr/bin/ipython
```

```
In [1]: import salt.client
```

```
In [2]: saltclient = salt.client.LocalClient()
```

```
In [3]: t3vmui01_disk_usage = saltclient.cmd('nodename:t3vmui01*', 'disk.usage', expr_form='grain' )
```

```
In [4]: print t3vmui01_disk_usage
```

```
{t3vmui01: {'/tmp': {'available': '1838008', '1K-blocks': '1975888', 'used': '35892', 'capacity': '2%',  
'filesystem': '/dev/md3'},  
'/var': {'available': '8704332', '1K-blocks': '9592544', 'used': '376572', 'capacity': '5%', 'filesystem':  
'/dev/md2'},  
'/boot': {'available': ...
```


Change the Minion state often means pushing files; I've defined a Python function to get the files from the Master

```
master # cat t3source.sls
#!pyobjects
```

```
def t3source( FILEPATH ):
    import os
    import re
    HOSTNAME      = os.uname()[1]                # t3ui01
    HOSTTYPE      = re.split( '[0-9][0-9]', HOSTNAME )[0] # t3ui
    OS            = __salt__['grains.get']('os_family') # RedHat
    OSREL         = __salt__['grains.get']('osrelease')[0] # 5
    SOURCE        = 'salt://' + 'OS/' + OS
    return [
        SOURCE + '/generic/files' + FILEPATH + '___' + HOSTNAME ,
        SOURCE + '/generic/files' + FILEPATH + '___' + HOSTTYPE ,
        SOURCE + '/generic/files' + FILEPATH ,
        SOURCE + '/' + OSREL + '/files' + FILEPATH + '___' + HOSTNAME ,
        SOURCE + '/' + OSREL + '/files' + FILEPATH + '___' + HOSTTYPE ,
        SOURCE + '/' + OSREL + '/files' + FILEPATH
    ]
```

The Minion files saved on the Master



SALTSTACK

```
master # find OS/RedHat | head -10
```

```
OS/RedHat
```

```
OS/RedHat/5
```

```
OS/RedHat/5/files
```

```
OS/RedHat/5/files/etc
```

```
OS/RedHat/5/files/etc/yum.conf.special
```

```
OS/RedHat/5/files/etc/quotatab__t3wn30
```

```
OS/RedHat/5/files/etc/quotatab__t3wn
```

```
OS/RedHat/5/files/etc/yum.conf.orig
```

```
OS/RedHat/5/files/etc/gmond.conf__t3vmui01
```

```
OS/RedHat/5/files/etc/ldap.conf__t3ui10
```

```
...
```


To change the Minions state we map 'states' (YAML or Python code) to Minions by using their Grain values

master # cat *top.sls*

base:

'os:ScientificLinux':

- match: grain

...

- states.cron
- states.smartd
- states.python
- states.nagios.common

'udev':

- match: nodegroup
- states.gcc
- states.snmpd
- states.nagios.check_linux_raid
- states.xrootd
- states.t3ui

master # cat states/*smartd*/init.sls

from salt://t3source.sls import **t3source**

with Pkg.installed("smartmontools"):

Service.running("smartd", enable=True)

with Service("smartd", "watch_in"):

File.managed('/etc/smartd.conf' ,
user='root',
group='root',
mode='0444',
source= **t3source**('/etc/smartd.conf'))

the 1st **with** means: if you can install "smartmontools" then enable the service "smartd"

the 2nd inner **with** means: service "smartd" must to be restarted if something enclosed by this with, in this case one File, changes.

Execution order will be generated respecting top-down *top.sls* BUT at runtime it also depends from the requires in the state logic

The previous states/**smartd**/init.sls, rewritten as YAML

smartmontools:

pkg:

- installed

smartd:

service:

- running
- require:
 - pkg: smartmontools
- watch:
 - file: /etc/smartd.conf

/etc/smartd.conf:

file.managed:

- source:
 - salt://OS/RedHat/5/files/etc/smartd.conf
- user: root
- group: root
- mode: 444

Observing the *top.sls* order before to run a Minion change

```
minion # salt-call state.show_lowstate --output=json
```

```
{
  "local": [
    {
      "group": "root",
      "name": "/tmp",
      "mode": "1777",
      "state": "file",
      "__id__": "/tmp",
      "fun": "directory",
      "__env__": "base",
      "__sls__": "states.tmp",
      "order": 10000,
      "user": "root"
    },
    {
      "group": "root",
      "name": "/etc/motd",
      ...
      "order": 10001,
```

```
    {
      "group": "root",
      "name": "/etc/inittab",
      "mode": "0644",
      "source": [
        "salt:"
        //OS/RedHat/5/files/etc/inittab__t3vmui01",
        "salt:"//OS/RedHat/5/files/etc/inittab__5.7",
        "salt:"//OS/RedHat/5/files/etc/inittab__t3vmui",
        "salt:"//OS/RedHat/5/files/etc/inittab"
      ],
      "state": "file",
      "__id__": "/etc/inittab",
      "fun": "managed",
      "__env__": "base",
      "__sls__": "states.general_etc",
      "order": 10002,
      "user": "root"
    },
```

If something fails during the Minion change should SaltStack go ahead, or immediately stop ?

Normally, when a state fails SaltStack continues to execute the remainder of the defined states and will only refuse to execute states that require the failed state.

But the situation may exist, where you would want all state execution to stop if a single state execution fails. The capability to do this is called **failing hard**.

- **failing hard** can be enforced globally (it remembers the Kickstart behaviour)
- or state by state
- because of the insane amount of time that I'm investing designing/testing states it's really unexpected the failure of a Minion change, so I want to immediately stop everything and double check !
- This could lead to write less 'requires' logic in my states and be heavily bounded to the order dictated by the *top.sls* file, but it's my duty to write solid states.

```
master # cat /etc/salt/master.d/failhard.conf
failhard: True
```

<http://docs.saltstack.com/en/latest/ref/states/failhard.html>

Concluding (but there is much more to be said ...)



SALTSTACK

PROS:

- I can write my configurations in Python !
- Pushing configuration files, installing Pkgs or restarting Services is one of the SaltStack features, not the feature.
- Google, Rackspace, HP and others are already using it for their Clouds.
- Minions can also:
 - Send events to each other (like distributed **inotify**)
 - Publish live data to each other (users connected, load, ...)
 - Send their output to MongoDB, MySQL, Postgres and many others.

CONS:

- New, the Doc is a bit chaotic and it took me a lot to see the big picture.
- Solaris support is several versions behind the Linux version.
- It requires to install and turn on yet another agent on the Minion ; competitors like [Ansible](#) just use SSH (btw it also uses YAML, it's also written in Python)