
Google ganeti

Ganeti

Ganeti Core Team - Google
LISA '13 - 5 Nov 2013

Automating Ganeti

Automating Aspects of Ganeti administration

- Guido Trotter <ultrotter@google.com>
- Helga Velroyen <helgav@google.com>

Latest version of these slides

Please find the latest version of these slides at:

https://code.google.com/p/ganeti/wiki/LISA2013

Programatic control of Ganeti

- Ganeti is all about automating the complex.
- You can write your own automation to control Ganeti.

Use bash for small scripts

Example: Find instances that are not using all the RAM allocated to them:

BASH

```
#!/bin/bash

ITEMS=$(gnt-instance list -o name,oper_ram,be/memory | awk '$2 != $3')
for i in $ITEMS ; do
    echo 'Why u no use your RAM,' $i '?'
done
```

- `list` is faster, and easier to parse, than `info`
- `gnt-*` commands don't return until the action is complete.
- Add `--submit` if waiting is not required.
- Submit long-running jobs with `--priority=low`

RAPI

- RAPI is the Remote API.
- (not to be confused with the API used between masterd and noded)
- It is RESTful
- Client library hides all the details.
- You just need the cluster name and (for write access) credentials.
- http://docs.ganeti.org/ganeti/current/html/rapi.html

Python Examples (1)

Read only requires no password:

```python
import ganeti_rapi_client as grc

rapi = grc.GanetiRapiClient('cluster1.example.com')

print rapi.GetInfo()
print rapi.GetInstances(bulk=True)
```

Tip: Results are often long. Make them readable with pprint:

```python
import pprint

pp = pprint.PrettyPrinter(indent=4).pprint
census = rapi.GetInstances(bulk=True)
pp(census)
```

Python Examples (2)

Read/Write requires credentials:

PYTHON

```python
import ganeti_rapi_client as grc

rapi = grc.GanetiRapiClient('cluster1.example.com')
rapi = grc.GanetiRapiClient(
    'cluster1', username='USERNAME', password='PASSWORD')

# Now "write" commands will work:
rapi.AddClusterTags(tags=['heuer'])
```

ProTip: Your cluster is alive

Bad: Things could change between queries:

```bash
gnt-instance list -F 'pnode == "gnta1"'
read -p 'Shut these down? ' ANS
if [[ $ANS == 'y' ]]; then
  gnt-instance list -F 'pnode == "gnta1"' -o name \
      --no-headings |
      xargs -n1 gnt-instance shutdown
fi
```

Good: Make list and work from it:

```bash
L=$(gnt-instance list -F 'pnode == "gnta1"' -o name --no-headings)
echo $L ; read -p 'Shut these down? ' ANS
if [[ $ANS == 'y' ]]; then
  echo $L | xargs -n1 gnt-instance shutdown -f
fi
```

Things to automate

- Adding instances of various types.
  - To insure all parameters are correct
- Periodic rebalances
  - Check to see if sufficiently unbalanced first
- Detect/fix DRBD issues
  - Find instances in degrated mode, stuck replication, etc.
  - Have a look at `harep`
- Workflow for evacuating a node:
  - remove from monitoring system
  - evacuate primaries and secondaries
  - check to see if evacuation complete
  - print that it is safe to power off node for maintenance
- Configuring a node

Automating node configuration

- Configure nodes consistantly:
  - package versions,
  - configuration files,
  - network configuration
- Ganeti runs smoother: Fewer "UFO" problems.
- Easier to administer: Less to remember.

Automate node configuration to achieve consistency.

```
"A foolish consistency is the hobgoblin of little minds."
--- Ralph Waldo Emerson

"Don't be a fool, configure all nodes consistently."
--- Guido and Tom
```

General strategy

- Use PXE to install a "base OS"
- Let installer partition the disks
- Use CfEngine, Puppet or Chef to configure the host

Puppet Tip 1:

Install specific version of a package, not 'latest':

- Reduces "surprise" upgrades in depot.
- Required for a "DEV -> QA -> PRODUCTION" strategy

PUPPET

```
package {
  'xen-hypervisor-4.0-amd64': ensure => '4.0.1-5.2';
  'ganeti2': ensure => '2.6.0-1';
}
```

Puppet Tip 2:

Add-ons like Augeas can edit complex configuration files:

PUPPET

```
augeas{"grup_ganeti_settings" :
  context => '/files/etc/default/grub',
  changes => [
    'set GRUB_DISABLE_OS_PROBER true',
    'set GRUB_CMDLINE_XEN_DEFAULT \'"dom0_mem=512M"\'',
  ]
}
```

Latest release understands the LISP-like format of `xend-config.sxp` and much, much more.

Puppet Tip 3:

`/etc/network/interfaces` can be generated by template...

PUPPET

```
$primary_interface_name = ...
$primary_interface_ip = ...
$replication_interface_name = ...
$replication_interface_ip = ...

file {
    path    => "/etc/network/interfaces",
    owner   => root,
    group   => root,
    mode    => 644,
    content => template('interfaces-2nic.erb),
}
```

Puppet Tip 3b:

...or use Augeas to edit `/etc/network/interfaces` in place:

PUPPET

```
augeas { "eth1":
  context => "/files/etc/network/interfaces",
  changes => [
    "set auto[child::1 = 'eth1']/1 eth1",
    "set iface[. = 'eth1'] eth1",
    "set iface[. = 'eth1']/family inet",
    "set iface[. = 'eth1']/method dhcp",
  ],
}
```

Autorepair (harep)

Before Ganeti 2.8, there was no self-repair:

- DRBD instance is broken
  - manually fail it over
  - trigger a disk replacement
- Plain instance is broken
  - Manually recreate disk(s) and reinstall

Harep

- The Ganeti autorepair tool
- Available since Ganeti 2.8
- Meant to be run regularly using cron
- Admin can allow/disallow specific repairs
- Design Doc: `doc/design-autorepair.rst`
  - Includes detailed description of all the intermediate tags used internally

Controlling autorepair

- Harep is controlled through tags
- `ganeti:watcher:autorepair:<type>`
  - instance/nodegroup/cluster
  - What kind of repair allowed? (Sorted, more risky includes less risky)
    - `fix-storage:` disk replacement or fix the backend without affecting the instance itself (broken drbd secondary)
    - `migrate:` allow instance migration
    - `failover:` allow instance reboot on the secondary
    - `reinstall:` allow disks to be recreated and the instance to be reinstalled

Risks

- `fix-storage:` data loss if something is wrong on the primary but the secondary was somehow recoverable
- `migrate:` can cause instance crash (bugs)
- `failover:` loses the running state
- `reinstall:` data loss

Preventing autorepair

- Blocking a few repairs is easier than changing all the enabled ones
- `repair:suspended`
  - prevents an instance from being touched
  - can specify an expiration timestamp

How does it work?

- Multiple states for instances
  - Healthy
  - Suspended
  - Needs repair, repair disallowed
  - Pending repair
  - Failed
- Every run of harep
  - updates the tags
  - submits jobs

The result

`ganeti:watcher:autorepair:result:<type>:<id>:<timestamp>:<result>:<jobs>`

- A `autorepair:result` tag is left on the repaired instance
- <repair>
  - success
  - failure
  - enoperm (=blocked by policies)

Thank You!

Questions?

Survey at https://www.usenix.org/lisa13/training/survey