Google ganeti

Ganeti Advanced Workshop

Ganeti Core Team - Google
GanetiCon 2013 - 3 Sept 2013

Monitoring Daemon

Michele Tartara <mtartara@google.com>

Once upon a time...

- there was no monitoring support in ganeti
- monitoring the status of the cluster was hard
  - Healthy?
  - Unhealty?
- difficult to correlate hardware info with instances info
  (for an observer outside the cluster)
  - Instance <--> Logical volume ?
  - Instance <--> DRBD volume ?

What is the monitoring daemon?

Provides information:

- about the cluster state
- about the cluster health
  - automatically computed
- live
- read-only

What is NOT the monitoring daemon?

- A general-purpose monitoring system.
- Not meant to compete with Nagios, Pacemaker...
  - Integrate with them!
  - Provide them with easily parsable internal information

It just aims to monitor Ganeti and the related parts of the system

How is the monitoring daemon?

- HTTP daemon
- Replying to REST-like queries
  - Actually, GET only
- Providing JSON replies
  - Easy to parse in any language
  - Already used in all the rest of Ganeti
- Optional
- Dependent on Confd
- Built upon Haskell's Snap library

Where is the monitoring daemon?

- Running on every node
- Not:
  - Only master-candidates
  - Only VM-enabled

What info does it provide?

Now:

- instance status (Xen only)
- diskstats information
- LVM logical volumes information
- DRBD status information
- Node OS CPU load average

Soon(-ish):

- instance status (KVM)
- Ganeti daemons status
- Hypervisor resources
- Node OS resources report

Data collectors

- provide data to the deamon
- one collector, one report
- one collector, one topic

Two kinds of collectors (I)

- Performance reporting
  - only provide data "as is"
  - no interpretation

Two kinds of collectors (II)

- Status reporting
  - provide status evaluation
    - healthy
    - being auto-fixed
    - unknown
    - broken
  - hide data
    - informed status decisions require deep internals knowledge
    - to prevent meddling when auto-fixing
    - verbose mode
      - Not implemented yet
      - currently always-on

The query response

- JSON
- one report per collector, in a list

```json
[
  {
  ... collector report
  },
  {
  ... collector report
  }
]
```

- common structure for all the reports
- specific fields for each collector

The report format (I)

JSON

```json
{
  "name" : "TheCollectorIdentifier",
  "version" : "1.2",
  "format_version" : 1,
  "timestamp" : 1351607182000000000,
  "category" : null,
  "kind" : 0,
  "data" : { "plugin_specific_data" : "go_here" }
}
```

- `name:` the name of the plugin. Unique string.
- `version:` the version of the plugin. A string.
- `format_version:` the version of the `data` format of the plugin. Incremental integer.
- `timestamp:` when the report was produced. Nanoseconds. Can be zero-padded.

The report format (II)

JSON

```json
{
  "name" : "TheCollectorIdentifier",
  "version" : "1.2",
  "format_version" : 1,
  "timestamp" : 1351607182000000000,
  "category" : null,
  "kind" : 0,
  "data" : { "plugin_specific_data" : "go_here" }
}
```

- `category`: the category of the collector
  - storage, instance, daemon, hypervisor, "null"
  - can define a minimum set of prescribed fields
- `kind`: the kind of the collector
  - performance reporting (`kind = 0`)
  - status reporting (`kind = 1`, more to come)

The report format (III)

JSON

```json
{
  "name" : "TheCollectorIdentifier",
  "version" : "1.2",
  "format_version" : 1,
  "timestamp" : 1351607182000000000,
  "category" : null,
  "kind" : 0,
  "data" : { "plugin_specific_data" : "go_here" }
}
```

- `data:` the collected data
  - free format
  - restrictions introduced by `category` and `kind`

Status reporting collectors: report

They introduce a mandatory part inside the `data` section.

```json
"data" : {
  ...
  "status" : {
    "code" : <value>
    "message: "some summary goes here"
  }
}
```

- `<value>`: by increasing criticality level
  - 0: working as intended
  - 1: temporarily wrong. Being auto-repaired
  - 2: unknown. Potentially dangerous state
  - 4: problems. External intervention required

Status reporting collectors: report

They introduce a mandatory part inside the `data` section.

JSON

```json
"data" : {
  ...
  "status" : {
    "code" : <value>
    "message: "some summary goes here"
  }
}
```

- `message`:
  - to better explain the reason of the status
  - optional (empty string) for codes 0 (ok) and 1 (auto-repair)
  - why could not be determined? (code 2, unknown)
  - what is wrong? (code 4, problems)

More goodies!

Enter `mon-collector`:

- quick 'n dirty CLI tool
- same collectors, same format
  - locally
- for quick checks by the sysadmin
- for local scripting
- shared code, different executables
  - works even if the daemon is not running
  - useful for offline-testing
- `$PREFIX/lib/ganeti/mon-collector <name>`
- `$PREFIX/lib/ganeti/tools/fmtjson`

What's where?

The development happens over time. Not everything is in every version.

- **2.7:**
  - mon-collector
  - DRBD data collector
- **2.8:**
  - monitoring daemon
- **2.9:**
  - logical volumes collector
  - instance status (XEN) collector
  - `/proc/diskstats`
- **2.10:**
  - CPU load (`/proc/stat`) collector. Thanks Spyros!

How to use the daemon?

- Accepts HTTP connections on `node.example.com:1815`
    - Not authenticated: read only
    - Just firewall, or bind on local address only
- GET requests to specific addresses
- Each address returns different info according to the API

The daemon API (I)

"Daemon, daemon on the port, would you send me a report?"

- `/`
  return the list of supported protocol version numbers
- `/1`
  the root of protocol version 1. Just returns `null`
- `/1/list/collectors`
  list of `(kind, category, name)` tuples, representing all the collectors

The daemon API (II)

"Daemon, daemon on the port, would you send me a report?"

- `/1/report/all`
  list of reports, one for each collector in the system
  - Will support `verbose=1`
- `/1/report/[category]/[collector_name]`
  report produced by `[collector_name]` belonging to `[category]`

Collector categories

Storage

- Gather data about the storage subsystem
- Different levels of granularity and abstraction
  - Physical disks, partitions, LVs, ...
- Always possible to trace back to the instance
  - To find out performance problems
  - Instance directly provided whenever possible
  - "References" between levels elsewhere
    - device name, LV name, ...
- No common fields

Collector categories

Hypervisor

- Hypervisor's view of system resources
- No such collector, so quite undefined
- Status reporting / Performance reporting?
- Fields?
  - Free/used memory, #CPUs, CPU average load, ...

Collector categories

Daemon

- Gather data about Ganeti's own deamons
- Help identifying memory leaks, crashes, high resource utilization, ...
- Status reporting collectors (`kind = 1`)
- One collector per daemon
- Common fields in verbose mode:
  - `memory`
  - `uptime`
  - `cpu_usage` (percentage)

Collector categories

Instance

- Status reporting collectors (`kind = 1`)
- Reports a global status, and a per-instance status
- List of instances, with hypervisor-independet fields
  - `name`
  - `uuid`
  - `admin_state`
  - `actual_state`
  - `uptime`
  - `mtime`
  - `state_reason`

Stateful data collectors (2.10)

Again, thanks Spyros

- Stateless collectors
  - Traditional ones: data collected at invocation time
- Stateful collectors
  - Collection function
    - Collects the data
    - Run regularly by the monitoring daemon
    - Stores data in the daemon itself (memory, for now. Collectors, behave!)
    - Daemon-wide constant collection timer
  - Reporting function
    - Receives the collected data
    - Elaborates and prints them
    - Can collect more

Conclusion

- More details and complete list of fields of the collectors in the design doc: `doc/design-monitoring-agent.rst`
- Future work:
  - Plugin system
  - KVM instance status collector
  - More collectors
  - Per-collector collection function timer

The reason trail

Michele Tartara <mtartara@google.com>

The reason trail

- Initially required for the instance status (Xen) collector
  - Why did the instance last change its status?
  - Not just a message, but a complete track of what happened
  - Decisions:
    - What format for expressing this?
    - Where to store the information?

What format?

List of triples `(source, reason, timestamp)`

```python
[("user", "Cleanup of unused instances", 1363088484000000000),
 ("gnt:client:gnt-instance", "stop", 1363088484020000000),
 ("gnt:opcode:shutdown", "job=1234;index=0", 1363088484026000000),
 ("gnt:daemon:noded:shutdown", "", 1363088484135000000)]
```

PYTHON

- `source:` the entity deciding to perform/forward the command. Free form, but the `gnt:` prefix is reserved
- `reason:` why the entity decided to perform the operation
- `timestamp:` timestamp since epoch, in nanoseconds

Where is it?

- Inside every opcode
- `op["reason"]` field
- Visible with `gnt-job info`

```
root@node1:~# gnt-job info 4
Job ID: 4
  ...
  Opcodes:
    OP_NODE_ADD
      ...
      Input fields:
        ...
        reason: ['gnt:client:gnt-node', 'add', 1377589019102071040],
                 ['gnt:opcode:op_node_add', 'job=4;index=0', 1377589019106505984]
```

How is it generated?

- Automatically, from RAPI/CLI down to opcode level
- Before opcode generation:
  - User message (now):
    - CLI: `--reason`
    - RAPI: `reason` parameter added to the request
  - Previous trail (future, if useful)
- After opcode's job execution:
  - Specialized usages and manual implementations
    - Instance state change reason (start, stop, reboot. Serialized on file)

Conclusion

- Since Ganeti 2.8!
- More information available in the design doc: `doc/design-reason-trail.rst`
- Future work:
  - Accept an initial trail as input

Configuration Daemon (ConfD)

Michele Tartara <mtartara@google.com>

Once upon a `t` ...

For `t < 2.1`

- Configuration only available on master candidates
- Few selected values replicated with Ssconf
  - Small pieces of config in text files on all the nodes
  - Doesn't scale
- Need a way to access config from other nodes
  - Scalable
  - No single point of failure (so, no RAPI)

Enters ConfD

- Provides information from `config.data`
- Read-only
- Distributed
  - Multiple daemons running on master candidates
  - Accessible from all the nodes through ConfD protocol
  - Resilient to failures
- Optional

What info does it provide?

Replies to simple queries:

- Ping
- Master IP
- Node role
- Node primary IP
- Master candidates primary IPs
- Instance IPs
- Node primary IP from Instance primary IP
- Node DRBD minors
- Node instances

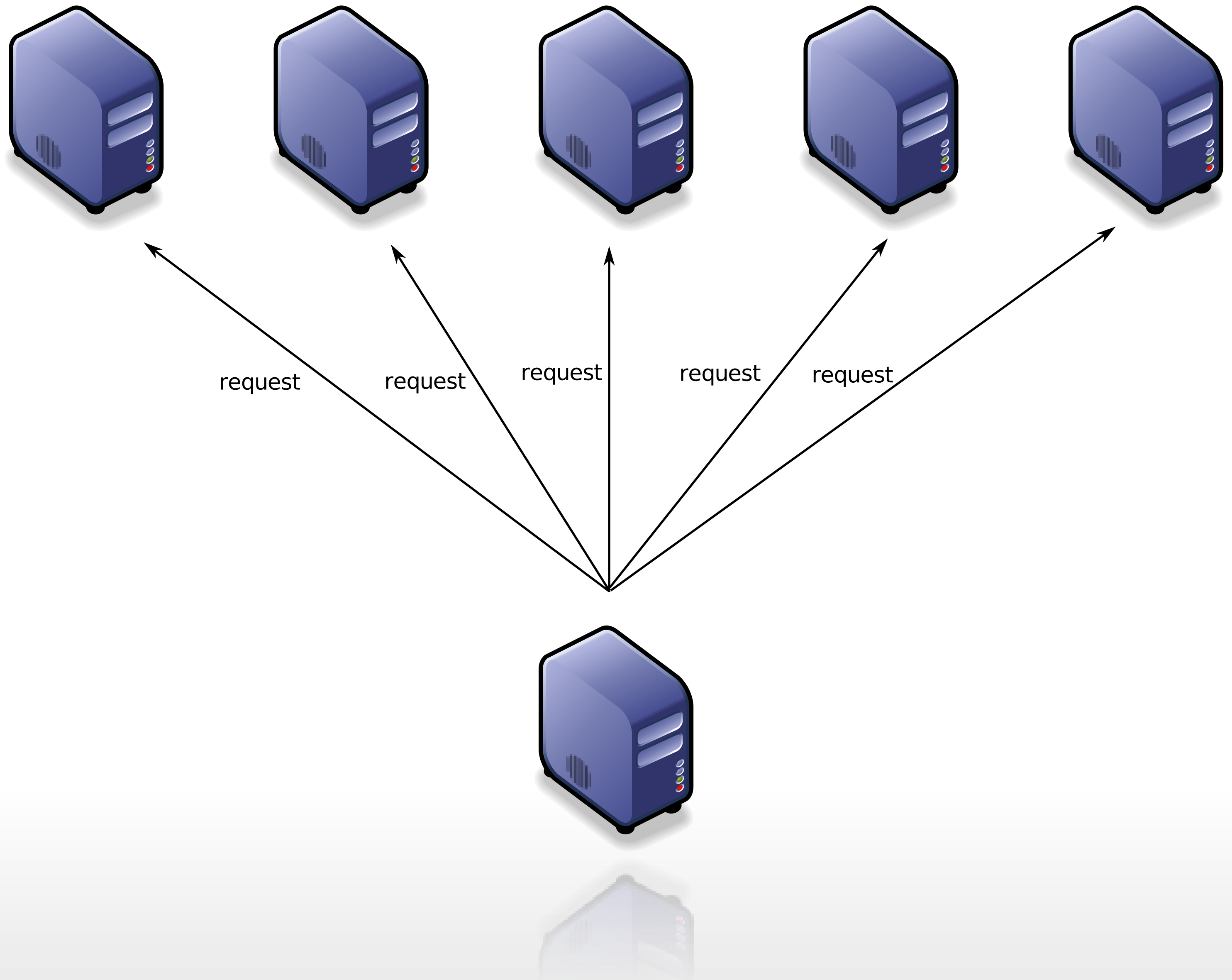
ConfD protocol

General description

- UDP (port 1814)
- keyed-Hash Message Authentication Code (HMAC) authentication
  - Pre-shared, cluster wide key
  - Generated at cluster-init
  - Root-only readable
- Timestamp
  - Checked (± 2.5 mins) to prevent replay attacks
  - Used as HMAC salt
- Queries made to any subset of master candidates
- Timeout
- Maximum number of expected replies

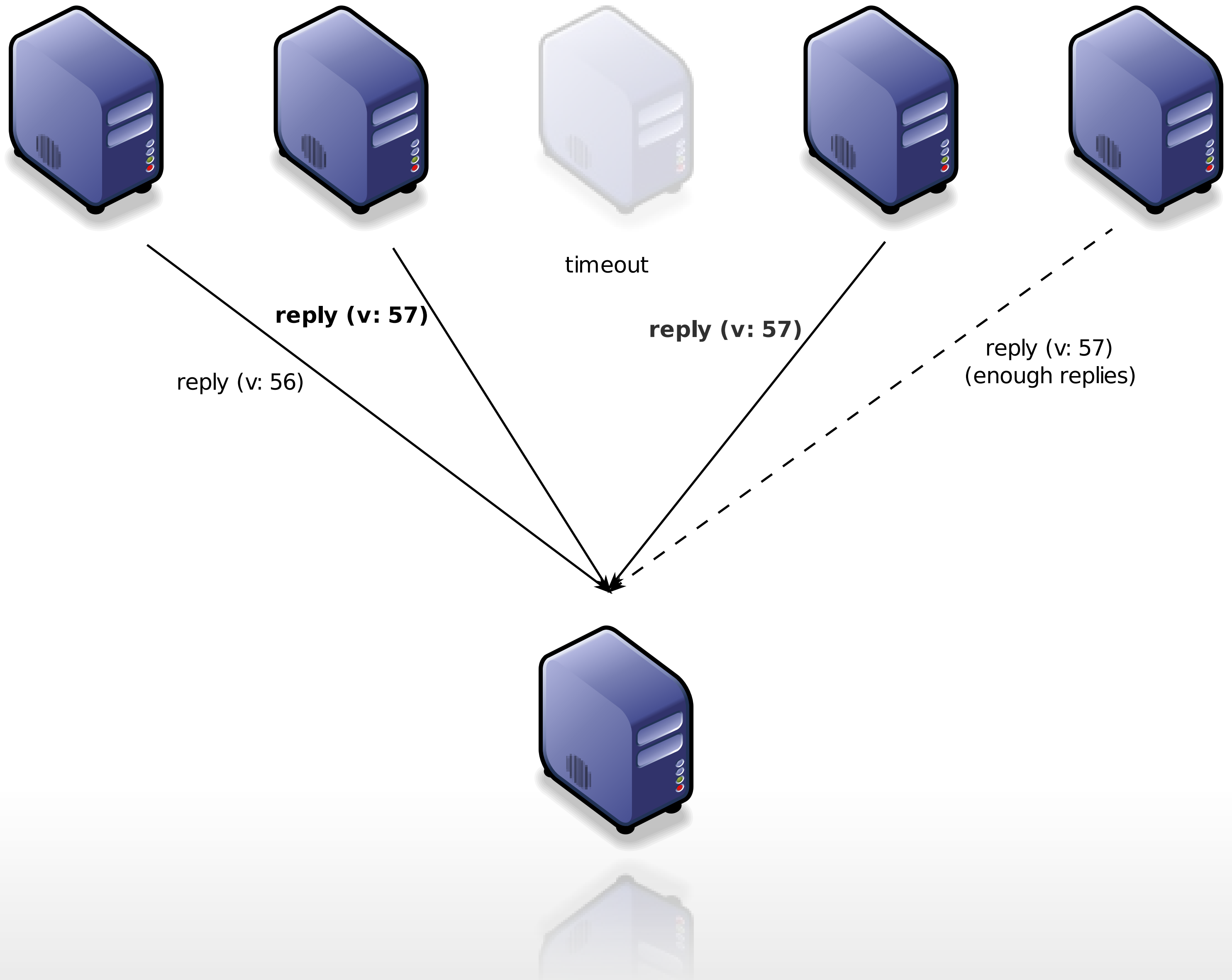
Confd protocol

Request/Reply



Confd protocol

Request/Reply



ConfD protocol

Request

CONFD

```
plj0{
  "msg": "{\"type\": 1,
            \"rsalt\": \"9aa6ce92-8336-11de-af38-001d093e835f\",
            \"protocol\": 1,
            \"query\": \"node1.example.com\"}\n",
  "salt": "1249637704",
  "hmac": "4a4139b2c3c5921f7e439469a0a45ad200aead0f"
}
```

- `plj0:` fourcc detailing the message content (PLain Json 0)
- `hmac:` HMAC signature of salt+msg with the cluster hmac key

ConfD protocol

Request

CONFD

```
plj0{
  "msg": "{\"type\": 1,
           \"rsalt\": \"9aa6ce92-8336-11de-af38-001d093e835f\",
           \"protocol\": 1,
           \"query\": \"node1.example.com\"}\n",
  "salt": "1249637704",
  "hmac": "4a4139b2c3c5921f7e439469a0a45ad200aead0f"
}
```

- `msg:` JSON-encoded query
  - `protocol:` ConfD protocol version (=1)
  - `type:` What to ask for (`CONFD_REQ_*` constants)
  - `query:` additional parameters
  - `rsalt:` response salt == UUID identifying the request

ConfD protocol

Reply

```confd
plj0{
  "msg": "{\"status\": 0,
            \"answer\": 0,
            \"serial\": 42,
            \"protocol\": 1}\n",
  "salt": "9aa6ce92-8336-11de-af38-001d093e835f",
  "hmac": "aaeccc0dff9328fdf7967cb600b6a80a6a9332af"
}
```

CONFD

- `salt:` the rsalt of the query
- `hmac:` hmac signature of salt+msg

ConfD protocol

Reply

CONFD

```
plj0{
  "msg": "{\"status\": 0,
           \"answer\": 0,
           \"serial\": 42,
           \"protocol\": 1}\n",
  "salt": "9aa6ce92-8336-11de-af38-001d093e835f",
  "hmac": "aaeccc0dff9328fdf7967cb600b6a80a6a9332af"
}
```

- `msg:` JSON-encoded answer
  - `protocol:` protocol version (=1)
  - `status:` 0=ok; 1=error
  - `answer:` query-specific reply
  - `serial:` version of `config.data`

Ready-made clients

The protocol is simple, but clients are simpler

- Ready to use ConfD clients
  - Python
    - `lib/confd/client.py`
  - Haskell
    - Since Ganeti 2.7
    - `src/Ganeti/ConfD/Client.hs`
    - `src/Ganeti/ConfD/ClientFunctions.hs`

Expanding ConfD capabilities

- Currently not so many queries are supported
- Easy to add new ones
  - Just add a new query type in the constants list
  - ...and extend the `buildResponse` function (`src/Ganeti/Confd/Server.hs` to reply to it in the appropriate way

Conclusion

- More info in `doc/design-2.1.rst`
- Future work
  - More queries can be easily added as needed
  - Management of the configuration (on master) moved to a separate daemon from masterd

Autorepair (harep)

Michele Tartara <mtartara@google.com>

Before Ganeti 2.8

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

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

Managing authorization conflicts

What if multiple autorepair tags act on an instance?

- In the same object: the least destructive takes precedence
- Across objects: the nearest tag wins
- Example:
    - cluster with I1 and I2
    - I1 has `failover`, the cluster has `fix-storage` and `reinstall`
    - Result: I1 `failover`, I2 `fix-storage`

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)

Conclusion

- More info `doc/design-autorepair.rst`
  - Includes detailed description of all the intermediate tags used internally

Cross-cluster instance migration

Guido Trotter <ultrotter@google.com>

Introduction

Instances can be moved between clusters that share a common secret.

- Operation available via the CLI or RAPI
- CLI tool uses RAPI, and can be seen as an example
- Data is transfered directly between clusters

Setup

Setup common secret and RAPI authentication

BASH

```bash
ssh root@cluster1 --> root@cluster1:~#
gnt-cluster renew-crypto --new-cluster-domain-secret
cat > /var/lib/ganeti/rapi/users <<EOF
mover testpwd write
EOF

# copy /var/lib/ganeti/cluster-domain-secret to the second cluster

ssh root@cluster2 --> root@cluster2:~#
gnt-cluster renew-crypto --cluster-domain-secret=path_to_domain_secret
# rapi access can be the same or different. in production use hashed passwords.
cat > /var/lib/ganeti/rapi/users <<EOF
mover testpwd write
EOF
```

Execute move

Can be run on a third party machine

BASH

```
PWDFILE=$(mktemp)
echo testpwd > $PWDFILE

# Note: --dst-* defaults to --src-* if not specified
/usr/lib/ganeti/tools/move-instance --verbose \
  --src-ca-file=rapi.pem --src-username=mover \
  --src-password-file=$PWDFILE \
  [--dest-instance-name=new_name --net=0:mac=generate] \
  --iallocator=hail cluster1 cluster2 instance.example.com
```

Bugs:

- Either--iallocator or nodes must be specified manually
- Move is slower than it ought to be

hspace

Klaus Aehlig <aehlig@google.com>

Introduction

Capacity planning

- How many more instances can I add to my cluster?
- Which resource will I run out first?

So simulate sequentially adding new machines

- until we run out of resources
- allocation done as with hail
- start with maximal size of an instance
  (as allowed by the policy)
- reduce size if we hit the limit for one resource

On a live cluster

Use Luxi backend to get live cluster data

BASH

```
# hspace -L
The cluster has 3 nodes and the following resources:
  MEM 196569, DSK 10215744, CPU 72, VCPU 288.
There are 2 initial instances on the cluster.
Tiered (initial size) instance spec is:
  MEM 1024, DSK 1048576, CPU 8, using disk template 'drbd'.
Tiered allocation results:
  -   4 instances of spec MEM 1024, DSK 1048576, CPU 8
  -   2 instances of spec MEM 1024, DSK 258304, CPU 8
  - most likely failure reason: FailDisk
  - initial cluster score: 1.92199260
  -   final cluster score: 2.03107472
  - memory usage efficiency:  3.26%
  -   disk usage efficiency: 92.27%
  -   vcpu usage efficiency: 18.40%
[...]
```

The simulation backend

One of the lesser known backends (hspace and hail)
Mainly for cluster planning

- Simulates an empty cluster with given data
- Format
  - allocation policy (p=preferred, a=last resort, u=unallocatable)
  - number of nodes (in this group)
  - disk space per node (in MiB)
  - ram (in MiB)
  - number of physikal CPUs
- use --simulate several times for more node groups

Planning a cluster

What if I bought 10 times more disks?

BASH

```bash
$ hspace --simulate=p,3,34052480,65523,24 \
> --disk-template=drbd --tiered-alloc=1048576,1024,8
The cluster has 3 nodes and the following resources:
  MEM 196569, DSK 102157440, CPU 72, VCPU 288.
There are no initial instances on the cluster.
Tiered (initial size) instance spec is:
  MEM 1024, DSK 1048576, CPU 8, using disk template 'drbd'.
Tiered allocation results:
  -  33 instances of spec MEM 1024, DSK 1048576, CPU 8
  -   3 instances of spec MEM 1024, DSK 1048576, CPU 7
  - most likely failure reason: FailCPU
  - initial cluster score: 0.00000000
  -   final cluster score: 0.00000000
  - memory usage efficiency: 18.75%
  -   disk usage efficiency: 73.90%
  -   vcpu usage efficiency: 100.00%
[...]
```

hroller

Klaus Aehlig <aehlig@google.com>

Introduction

When rebooting all nodes (e.g., kernel update),
there are several things to take care of.

- Don't reboot primary and secondary at the same time.
  Machine/disks might not come back after reboot.
- When doing live migration, have enough memory.
  No two nodes with primaries, that have the same secondary.
- When fully evacuating, plan for disk space.

Default

hroller suggests groups of nodes to be rebooted together.
By default, plan for live migration.

BASH

```
# hroller -L
'Node Reboot Groups'
node-00,node-10,node-20,node-30
node-01,node-11,node-21,node-31
```

Also possible to only avoid primary/secondary reboots (--offline-maintenance) or to plan complete node evacuation (--full-evacuation).

BASH

```
# hroller -L --full-evacuation
'Node Reboot Groups'
node-01,node-11
node-00,node-10
node-20,node-30
node-21,node-31
```

Moves

For the full evacuation, moves can also be shown (--print-moves).
Typically, together with --one-step-only.

BASH

```bash
# hroller -L --full-evacuation --print-moves --one-step-only
'First Reboot Group'
node-01
node-11
  inst-00 node-00 node-20
  inst-00 node-00 node-10
  inst-10 node-10 node-21
  inst-11 node-10 node-00
```

Tags

Nodes to be considered can also be selected by tags.
This allows reboots interleaved with other operations.

BASH

```bash
GROUP=`hroller --node-tags needsreboot --one-step-only --no-headers -L`
for node in $GROUP; do gnt-node modify -D yes $node; done
for node in $GROUP; do gnt-node migrate -f --submit $node; done
# ... wait for migrate jobs to finish
# reboot nodes in $GROUP
# verify...
for node in $GROUP; do gnt-node remove-tags $node needs-reboot; done
for node in $GROUP; do gnt-node modify -D no $node; done
hspace -L -X
```

Thank You!

Questions?