Monitoring Doc

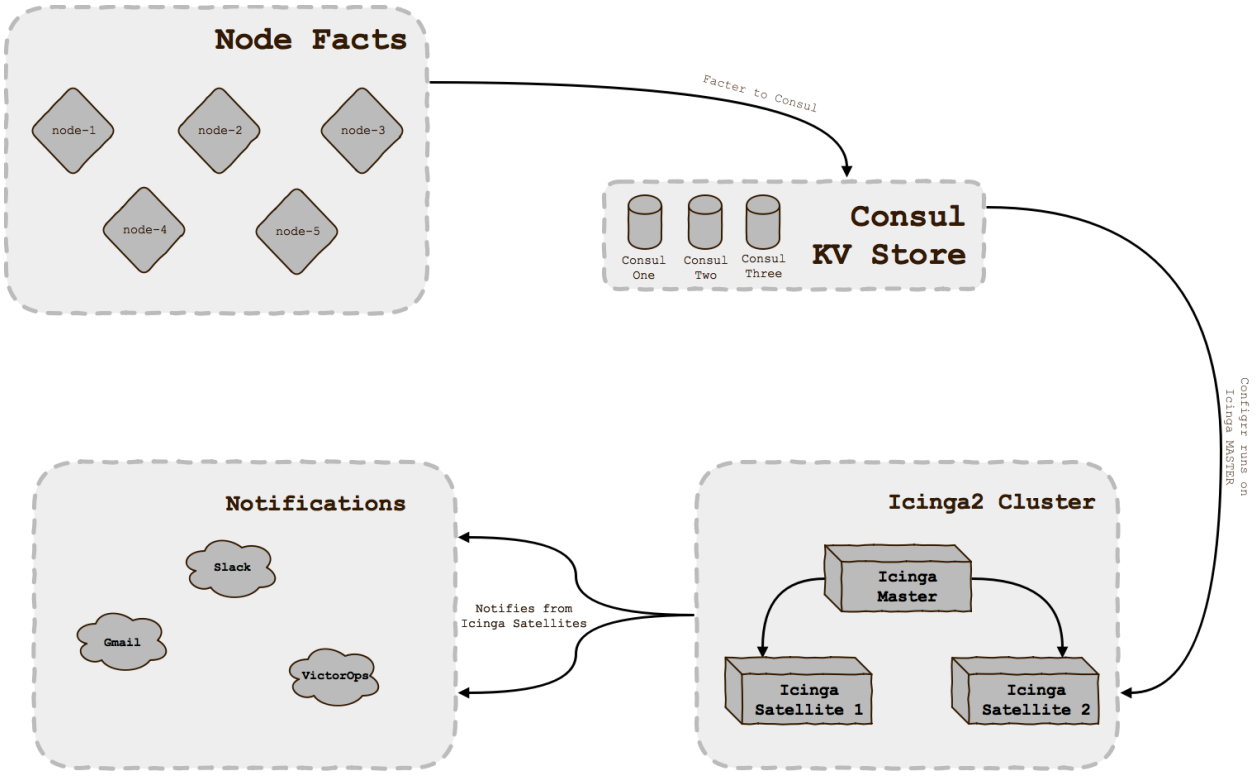
Overview

The current Nagios infrastructure has grown beyond its capabilities. Besides not being scalable, the version of Nagios is outdated and we think some of the bugs we are seeing are due to this. We will take this opportunity to find a replacement for Nagios that is actively supported and integrate with the rest of our infrastructure. We will also change the way configuration is generated to alleviate load off of Puppetservers. We will also explore the possibility of adding Jenkins tests before deploying the code.

Below are the alternatives for Nagios that were considered.

- Icinga2 (Complete rewrite, new syntax, conditionals)
- Shinken (Nagios rewritten in Python, same syntax)
- Sensu (Erlang/Ruby, JSON, RabbitMQ, Redis, high overhead)
- Nagios 4 (Same syntax, same scalability, or lack thereof)

Diagram



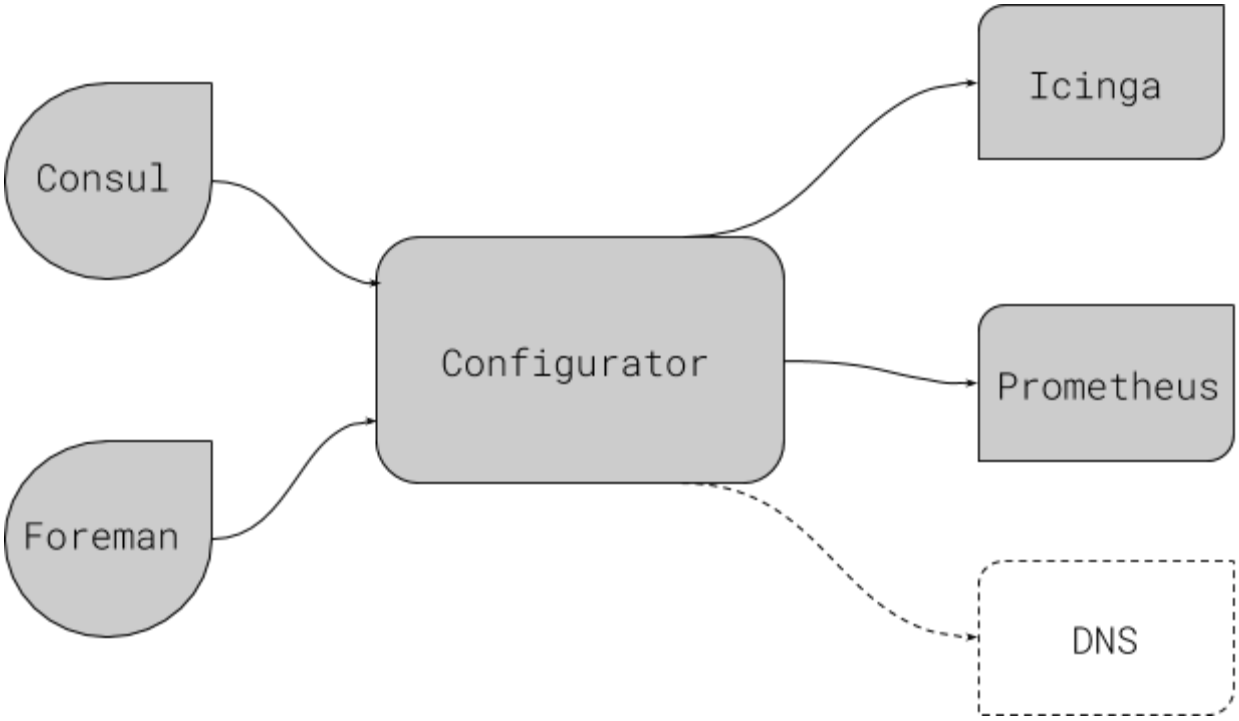
Config Generation

The config generation will be decoupled from Puppet since runs take forever to apply when the hosts, checks, and other resources are exported. In Icinga2 the config files could be generated in one of two ways:

- Meth 1: Have user commit config files directly to git and Icinga2 server can pull down the configs. This is similar to how we manage DNS zones currently.
  - Easy to setup.
  - Steep learning curve, prone to mistakes, user needs to be familiar with monitoring system syntax and quirks.

- Meth 2: Have user create a YAML file and have a lightweight wrapper fetch data from Foreman API. It will run via cron pull down the YAML files from git and generate the config based on the list of hosts from Foreman.
  - 100% self-serviceable; user does not need to know monitoring config syntax, just needs to update YAML files.
  - Easy to port data in case we have to switch monitoring systems in the future.
  - Can double up as config generation for other parts of the infra, such as DNS.
  - Not difficult to set up but does take time to write/test the wrapper.

Config Generation POC: I did a proof of concept on Meth #2. Lets dive a little deeper.



https://git.ccmteam.com/projects/SM/repos/configrr/browse

```
# ./configrr generate
Generating config for hosts...done.
Generating config for hostgroups...done.
```

Configrr: Is a simple wrapper that pulls data from API sources such as Foreman and Consul. In this example we mimic calling Foreman API. Note we do not actually call the Foreman API in this POC since the call takes over a minute to respond. Once in production the IO.read will be replaced with the actual call to Foreman/Consul.

After pulling in the data the hash is sanitized so only the key/values we actually use are available to us. Using the keys we can generate hostgroups based on the value. For instance if the key is *role* and value is *fe_mongo*, the hostgroup that is automagically generated would be *role_fe_mongo* (see *data* dir).

Templates: This dir holds the ERB templates that would render Icinga configuration. The sanitized hashes from Foreman and piped through to the ERB and the config is generated based on the template.

Config: This dir is a work in progress but pretty much anything that is hardcoded in Config_gen.rb that can be moved to a YAML configuration file will be put in this. This will include custom hostgroups, service checks, timeperiods, etc.

Data: The final rendered version of the template are stored in this dir. Usually this would be the config dir of the monitoring system. In the case of Icinga 2 this would be /etc/icinga2/conf.d/.

* The final wrapper would have proper exception handling, logging, etc.

Check config via Jenkins

By moving config to git we can automate checking the configuration even before it gets deployed. By using Jenkins, as long as the Icinga2 exec is available, it can run a check on the generated configuration to ensure it doesn't break Icinga2 when the code is pushed.

Performance Data

Icinga 2 has a built in InfluxDB exporter to export the performance data. Its enabled using the following commands:

```
# icinga2 feature enable perfdata
# icinga2 feature enable influxdb
```

InfluxDB Data Store

InfluxDB is the backend store for performance data. This would probably need to be setup in a cluster for high availability (some testing is still left to do),

Thruk Web UI

Thruk aggregates hosts and services in a web UI from multiple Icinga nodes. In essence only Icinga core needs to be installed on all the nodes and a central server can host Thruk.

Maintenance Mode

Icinga2 has concept of rules and conditionals, which are pretty powerful. You can do cool things like tag a node as *maintenance* and add

a rule to the Notification to not notify if a node is tagged with a certain tag, in this case maintenance.

```
object Host "too.cool.for.school" {
  check_command = "hostalive"
  address = "10.0.0.2"

  vars.notification_status = "maintenance"
}

object Host "legit.host" {
  check_command = "hostalive"
  address = "10.0.0.3"

  vars.notification_status = "active"
}

apply Notification "host-victorops" to Host {
  import "generic-notification"

  if (host.vars.notification_status == "maintenance") {
    command = "email-host-notification"
  } else {
    command = "victorops-host-notification"
  }

  assign where host.address
}
```

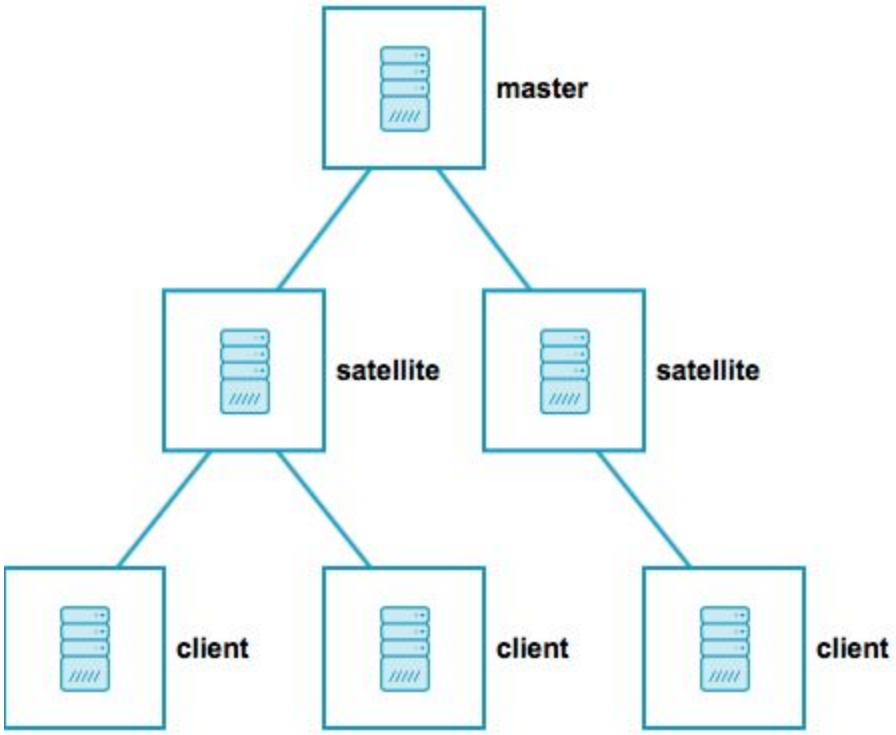
Host vs. Service checks

The latest Icinga 2 is a major improvement over Nagios 3 but it still has a few limitations in the way it operates. For example Icinga 2 is not designed to run checks for less than 5 mins intervals, it would tip over. For disk, memory, swap, BMC this duration is acceptable but Services on the other hand need sub-minute interval checks, 5 minutes is too long to detect a degradation in service.

The proposal here is to use Icinga2 for Host checks and Prometheus for Service checks.

Distributed cluster

Icinga2 has been designed from the ground up with HA and Load balancing in mind. The Icinga 2 cluster consists of 1 master node and 2 satellite nodes. The master node is responsible for accepting check results, sending notifications, etc. The satellite nodes would run the host checks, passively receive results from service checks, send results up to the master. The satellite pool can be expanded to add more nodes if required. The Master can also be designed to have HA but it needs MySQL as a backend. The satellite nodes will continue to operate and run checks for a little while if the master node goes down.



Client/Agent

While some checks can be run remotely via a port, API or SNMP, most advanced plugins require the plugin to be installed directly on the node. The locally installed agent will aggregate results from all plugins and send only the results up to the satellite nodes. Another advantage of running checks passively this way is the load for running the plugins will be dispersed among all the nodes rather than satellite nodes trying to do it all.