NYMMS Documentation

Release 0.2.6

Michael Barrett

June 06, 2016

Contents

NYMMS is a monitoring system written in python that takes influences from many of the existing monitoring systems. It aims to be easy to scale and extend.

CHAPTER 1

Demo

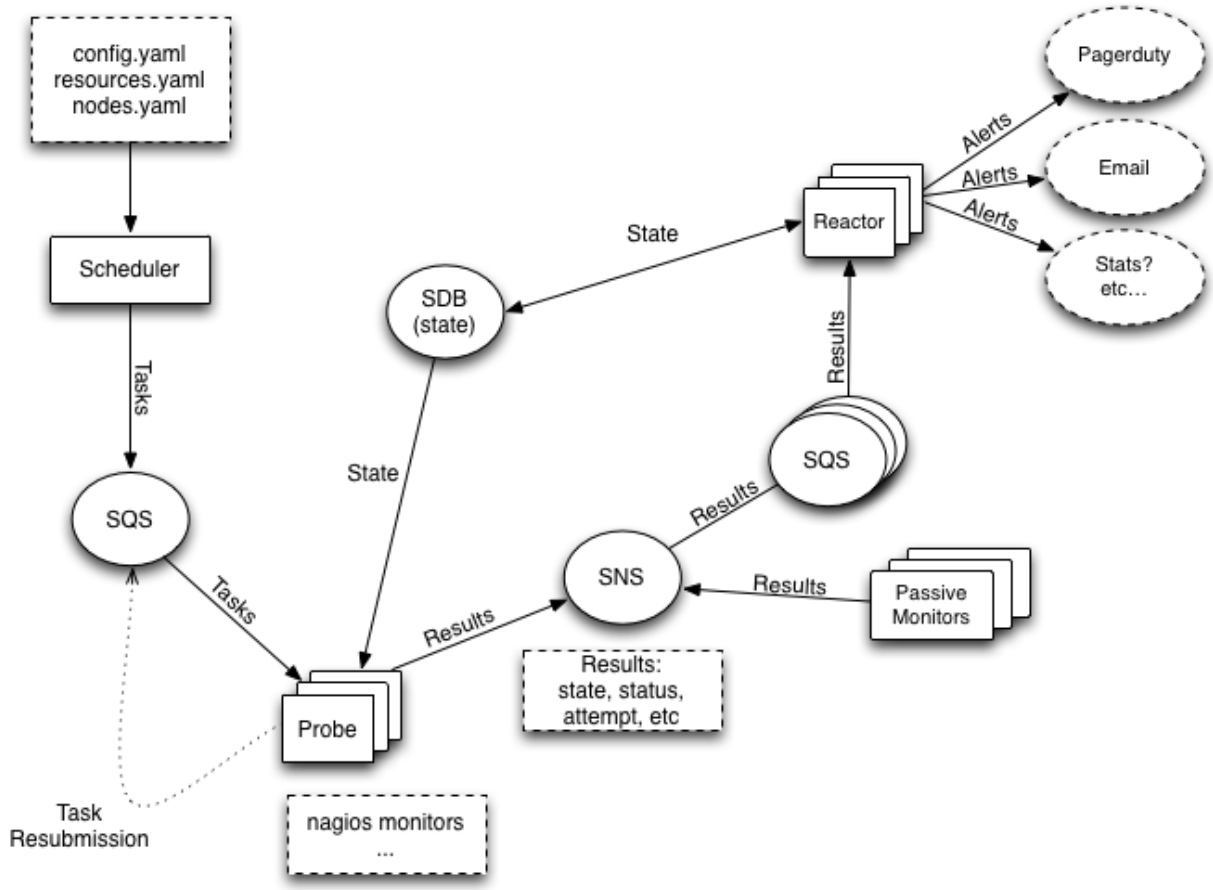
Before we get into the guts of NYMMS I'd like to mention that we build a demonstration Amazon AMI that comes up with a basic configuration for an all-in-one NYMMS host that runs all of the daemons. To get more information on how to use that, please visit Demo AMI

CHAPTER 2

Scaling

NYMMS intends to scale as easily as possible. It does so by separating out the work often handled in a monitoring system into multiple processes, and then handling communication between those processes with queues. None of this is revolutionary (Shinken broke the Nagios daemon up into many small pieces, and Sensu made heavy use of queues, and all of them are excellent monitoring systems that we take heavy influence from)- but I'm hoping to bring the two together in useful ways.

2.1 Architecture Diagram



2.2 The Daemons

nymms-scheduler: The daemon responsible for reading in the configuration, figuring out what it is you want to monitor and how you want to monitor those things, and then submitting tasks to the queue for probes.

nymms-probe: The daemon(s) responsible for reading from the task queue and taking those monitoring tasks and executing them. It sends along the results of those monitors to the results topic.

nymms-reactor: The daemon(s) that takes all the results, applies filters to them and then passes off the results that pass the filters onto their various handlers. Handler's can do just about anything with the results, from emailing people to triggering an incident in PagerDuty, to submitting stats to a stats system. Finally the reactor updates the state database with the result.

2.3 Communication

I've tried to keep the interface with the various communication channels simple and easily extendible. As of this writing the entire system is very AWS based. We make use of the following AWS services:

SQS: We use SQS as our general queue service. The scheduler passes tasks to the probes via SQS directly. The reactors read the results from the probes off SQS queues (note that the probes don't send results directly through SQS, which leads us to...)

SNS: Probes submit results into SNS topics, which then pass them onto the reactors' SQS queues. This allows a single result to be shared amongst multiple types of reactors, as well as allowing results to be sent to various other endpoints.

SDB: We use AWS SimpleDB to store state. This state database is written to by reactors when they receive results. It's read from by probes (to make sure we aren't beating a dead horse when something is down and has been down for some time) and by the reactors (to allow for logic regarding reacting to results that have changed state, or have been in a state for some length of time).

SES: We use AWS Simple Email Service in some reactor handlers in order to be able to easily send email.

Each of these services is used fairly lightly in most cases, so the charges should be minimal in almost all cases. The upside is that we currently do not require physical servers for any of these functions, which inevitably cost a significant amount to build and maintain.

In the future it should be fairly easy to convert these services to other systems (such as RabbitMQ, MongoDB, etc).

2.4 Other Details

Right now all monitors are active monitors - they are fired from the probes and contact other services via various protocols to determine if the service is in an okay state. Because of the design using the various queues however, it should be simple in the future to submit passive results. The reactors are very permissive in accepting data from just about any source just as long as it comes from their queue and it fits the correct dataformat.

As well we use a plugin format identical to the Nagios format. The benefit of this is that there is a vast wealth of nagios plugins out there, and they can be used as is with NYMMS. In the future we may come up with other plugin formats, but we haven't had a reason to so far.

CHAPTER 3

Contents

3.1 Configuration

The default configuration language for NYMMS is written in YAML. For the most part it follows the YAML standard. It has one main addition, the !include macro.

!include can be used to include another file in a given file. This is useful when you have a main config file (say *nodes.yaml*) but want to allow external programs to provide more config (say in `/etc/nymms/nodes/*.yaml`).

In that specific example you'd put the following in the yaml file where you want the files included:

```
!include /etc/nymms/nodes/*.yaml
```

3.1.1 config.yaml

The config.yaml file is the main configuration for all of the daemons and scripts in NYMMS.

You can see an example by expanding the code block below.

monitor_timeout This represents the default amount of time, in seconds, each monitor is given before it times out. *Type:* Integer. *Default:* 30

resources This points to the filesystem location of the resources config (see *resources.yaml*). *Type:* String, file location. *Default:* `/etc/nymms/resources.yaml`

region The AWS region used by the various daemons. *Type:* String, AWS Region. *Default:* us-east-1

state_domain The SDB domain used for storing state. *Type:* String. *Default:* nymms_state

tasks_queue The name of the SQS queue used for distributing tasks. *Type:* String. *Default:* nymms_tasks

results_topic The name of the SNS topic where results are sent. *Type:* String. *Default:* nymms_results

private_context_file The location of the private context file (see *private.yaml*). *Type:* String, file location. *Default:* `/etc/nymms/private.yaml`

task_expiration If a task is found by a probe, and it is older than this time in seconds, then the probe will throw it away. *Type:* Integer. *Default:* 600

probe This is a dictionary where probe specific configuration goes. *Type:* Dictionary.

max_retries The maximum amount of times the probe will retry a monitor that is in a non-OK state. *Type:* Integer. *Default:* 2

queue_wait_time The amount of time the probe will wait for a task to appear in the tasks_queue. AWS SQS only allows this to be a maximum of 20 seconds. In most cases, the default should be fine. *Type:* Integer. *Default:* 20

retry_delay The amount of time in seconds that a probe will delay retries on non-OK, non-HARD monitors. This allows you to quickly retry monitors that are supposed to be failing, to verify that there is an actual issue. *Type:* Integer. *Default:* 30

reactor This is a dictionary where reactor specific configuration goes. *Type:* Dictionary

handler_config_path The directory where *Reactor Handlers* specific configurations are found. *Type:* String. *Default:* `/etc/nymms/handlers`

queue_name The name of the SQS queue where reactions will be found. *Type:* String. *Default:* reactor_queue

queue_wait_time The amount of time the probe will wait for a result to appear in the queue named in reactor.queue_name. AWS SQS only allows this to be a maximum of 20 seconds. In most cases, the default should be fine. *Type:* Integer. *Default:* 20

visibility_timeout The amount of time (in seconds) that a message will disappear from the SQS reactor queue (defined in reactor.queue_name above) when it is picked up by a reactor. If the reactor doesn't finish it's work and delete the message within this amount of time, the message will re-appear in the queue. This allows the reactions to survive reactor crashes and the like. *Type:* Integer. *Default:* 30

scheduler This is a dictionary where reactor specific configuration goes. *Type:* Dictionary

interval How often, in seconds, the scheduler will schedule tasks. *Type:* Integer. *Default:* 300

backend The dot-separated class path to use for the backend. The backend is what is used to find nodes that need to be monitored. *Type:* String. *Default:* nymms.scheduler.backends.yaml_backend.YamlBackend

backend_args Any configuration args that the scheduler.backend above needs. *Type:* Dictionary

path This is used by the YamlBackend, which is the default. This gives the name of the yaml file with node definitions that the YamlBackend uses. *Type:* String. *Default:* `/etc/nymms/nodes.yaml`

lock_backend The backend used for locking multiple schedulers. Currently only SDB is available. *Type:* String. *Default:* SDB

lock_args Any configuration args that the scheduler.lock_backend needs. *Type:* Dictionary.

duration How long, in seconds, the scheduler will keep the lock for. *Type:* Integer. *Default:* 360

domain_name The SDB domain name where locks are stored. *Type:* String. *Default:* nymms_locks

lock_name The name of the lock. *Type:* String. *Default:* scheduler_lock

suppress These are the config settings used by the suppression system. *Type:* Dictionary.

domain The SDB domain where suppressions will be stored. *Type:* String. *Default:* nymms_suppress

cache_timeout The amount of time, in seconds, to keep suppressions cached. *Type:* Integer. *Default:* 60

3.1.2 resources.yaml

The resources.yaml file is where you define your commands, monitors and monitoring groups.

commands Commands are where you define the commands that will be used for monitoring services. The main config for each command is the *command_string*, which is a templatized string that defines the command line to a command line executable.

monitors Monitors are specific instances of commands, allowing you to fill in templated variables in the command used. This allows your commands to be fairly generic and easily re-usable.

monitoring groups Monitoring groups are used to tie monitors to individual nodes. It also lets you add some monitoring group specific variables that can be used in commands templates and other places.

Config Options

commands A dictionary of commands, the key of each is a unique name for the command, and the value is another dictionary with the commands configuration. Other than the *command_string* config option, you can specify any others you like - they will be accessible in the template of the *command_string* itself. *Type:* Dictionary.

command_string A command line string using Jinja's variable syntax. (ie: {{variable}}). *Type:* String.

other configs You can specify as many other key/value entries as you like. They will be useable as variables in the *command_string* itself. Often times the values set here will be used as defaults for the command, provided the variable isn't set anywhere else (such as on the monitor, or the node).

monitors A dictionary of monitors, each of which calls a command defined above. The key of each entry is the name of the monitor, the value is another dictionary which contains configuration values for that monitor. *Type:* Dictionary

command The name of a command defined in the resources file. This is the command that will be called for this monitor. *Type:* String.

monitoring_groups A list of monitoring groups that this monitor is a part of. This is how you tie monitors to nodes - every monitor that is attached to a monitoring_group will be ran against every node that is attached to that monitoring_group.

other configs You can specify as many other key/value entries as you like for each monitor. They will be useable as variables in the template strings used in the command for this monitor.

monitoring_groups A dictionary of monitoring groups which tie together monitors and nodes. The keys of the dictionary are the monitoring_groups names, while the values are any extra config you want to put into the command context. Often times the values will be blank (see the example).

3.1.3 private.yaml

The private.yaml file is used to give context variables that can be used in various monitors, but which are not included when the tasks and results are sent over the wire. Largely these are used for things like passwords that are needed by monitors.

The variables that are provided by private.yaml need to be prepended by *__private.* when referring to them in templates. For example, if you have a private variable called *db_password* you would refer to it as *__private.db_password* in templates.

The contents of the private.yaml are simple key/value pairs.

3.1.4 nodes.yaml

The nodes.yaml file is the file used by default by the YamlBackend, which is used by the scheduler to figure out what nodes (instances, hosts, etc) need to be monitored. It's a dictionary of node entries - each entry's key is the name of the node. The value of each entry is a dictionary with the following options:

address The network address of the node. This can be an ip address, or a hostname. If no address is provided, then it is assumed that the name of the node entry is the address. *Type:* String. *Default:* The node entry name.

monitoring_groups A list of monitoring groups (as defined in resources.yaml) that this node is part of. Every monitor that is attached to a monitoring group will be applied to every node in the monitoring group. *Type:* List.

realm The realm this node is a part of. See the realms documentation.

3.1.5 Reactor Handlers

3.2 Demo AMI

In order to give people something easy to start playing with (and to alleviate my shame in not having amazing documentation yet) I've gone ahead and started creating Demo AMIs in Amazon AWS. These AMIs come up with a complete, all-in-one (ie: all daemons) instance that has a very basic configuration that can be used to play with NYMMS and get used to the system.

Currently the AMIs are only being built in **us-west-2 (ie: oregon)** but if you have interest in running the AMI elsewhere contact me and I'll see about spinning one up for you.

You can find the AMIs by searching in the EC2 console in **us-west-2** for **nymms**. The AMIs are named with a timestamp like so:

nymms-ubuntu-precise-20131014-215959

Once you launch the AMI (I suggest using an m1.medium, though it MAY be possible to use an m1.small) you'll need to provide it with the correct access to the various AWS services (SQS, SNS, SES, SDB) that NYMMS makes use of.

This can be done one of two ways:

- You can create an instance role with the appropriate permissions (given below) and assign the instance to it.
- You can create an IAM user and assign the appropriate permissions then take their API credentials and put them in **/etc/default/nymms-common**

The first way is the more secure, but the second is the easiest. Here's an example permission policy that should work:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "ses:GetSendQuota",
        "ses:SendEmail"
      ],
      "Sid": "NymmsSESAccess",
      "Resource": [
        "*"
      ],
      "Effect": "Allow"
    },
    {
      "Action": [
        "sns:ConfirmSubscription",
        "sns:CreateTopic",
        "sns:DeleteTopic",
        "sns:GetTopicAttributes",
        "sns:ListSubscriptions",
        "sns:ListSubscriptionsByTopic",
        "sns:ListTopics",
        "sns:Publish",
        "sns:SetTopicAttributes",
```

```
        "sns:Subscribe",
        "sns:Unsubscribe"
      ],
      "Sid": "NymmsSNSAccess",
      "Resource": [
        "*"
      ],
      "Effect": "Allow"
    },
    {
      "Action": [
        "sqs:ChangeMessageVisibility",
        "sqs:CreateQueue",
        "sqs:DeleteMessage",
        "sqs:DeleteQueue",
        "sqs:GetQueueAttributes",
        "sqs:GetQueueUrl",
        "sqs:ListQueues",
        "sqs:ReceiveMessage",
        "sqs:SendMessage",
        "sqs:SetQueueAttributes"
      ],
      "Sid": "NymmsSQSAccess",
      "Resource": [
        "*",
      ],
      "Effect": "Allow"
    },
    {
      "Action": [
        "sdb:*"
      ],
      "Sid": "NymmsSDBAccess",
      "Resource": [
        "*"
      ],
      "Effect": "Allow"
    }
  ]
}
```

Once you've done all that you need to restart each of the three nymms daemons via upstart so that they can read their new credentials:

```
# restart nymms-reactor
# restart nymms-probe
# restart nymms-scheduler
```

If all went well (you can tell by checking out the individual daemon logs in **/var/log/upstart/**) you should start to see the results of the very basic monitors in **/var/log/nymms/reactor.log**.

You can find all of the configuration in **/etc/nymms**.

Let me know if you have any questions or run into any issues bringing up the AMI/services.

3.3 Getting Started with NYMMS

This tutorial will walk you through installing and configuring NYMMS. If you'd quickly like to start a NYMMS system to play with yourself, please see the Demo AMI documentation.

This tutorial assumes basic understanding of Amazon Web Services. You will either need to understand how to launch an instance with an instance profile with the appropriate permissions (see below) or you will need the `Access Key ID` and `Secret Access Key` for a user with the appropriate permissions.

3.3.1 Installing NYMMS

On Ubuntu

Maintaining the Ubuntu packages proved to be difficult after NYMMS started using multiple third party python packages. Because of that, we no longer maintain the Ubuntu packages. Instead you should use the docker images (see below)

Using Docker

A docker image is provided that can be used to run any of the daemons used in NYMMS. It can be pulled from *phobologic/nymms*. To run the daemons, you can launch them with the following command:

> docker run -e "AWS_ACCESS_KEY_ID=<AWS_ACCESS_KEY_ID>" -e "AWS_SECRET_ACCESS_KEY=<AWS_SECRET_ACCESS_KEY>" –rm -it phobologic/nymms:latest /[scheduler|probe|reactor] <OPTIONAL_ARGS>

For example, to run the scheduler (with verbose logging, the -v) you can run:

> docker run –rm -it phobologic/nymms:latest /scheduler -v

You can also set the *AWS_ACCESS_KEY_ID & AWS_SECRET_ACCESS_KEY* in a file, and then use *–env-file* rather than specifying the variables on the command line. Optionally, if you are running on a host in EC2 that has an IAM profile with all the necessary permissions, you do not need to specify the keys at all.

The docker container has the example config, which just checks that *www.google.com* is alive. It only has a single reactor handler enabled, the log handler, which logs to */var/log/nymms/reactor.log*.

To use the docker container with your own configs, you should put them in a directory, then mount it as a volume when you run the containers. If you put the configs in the directory */etc/nymms* on the host, you should run the container like this:

> docker run -v /etc/nymms:/etc/nymms:ro –rm -it phobologic/nymms:latest /scheduler -v

Using PIP

Since NYMMS is written in python I've also published it to PyPI. You can install it with pip by running:

```
pip install nymms
```

Warning: The python library does not come with startup scripts, though it does install the three daemon scripts in system directories. You should work on your own startup scripts for the OS you are using.

Installing From Source

You can also install from the latest source repo:

```
git clone https://github.com/cloudtools/nymms.git
cd nymms
python setup.py install
```

Warning: The python library does not come with startup scripts, though it does install the three daemon scripts in system directories. You should work on your own startup scripts for the OS you are using.

Using Virtual Environments

Another common way to install NYMMS is to use a virtualenv which provides isolated environments. This is also useful if you want to play with NYMMS but do not want to (or do not have the permissions to) install it as root. First install the virtualenv Python package:

```
pip install virtualenv
```

Next you'll need to create a virtual environment to work in with the newly installed virtualenv command and specifying a directory where you want the virtualenv to be created:

```
mkdir ~/.virtualenvs
virtualenv ~/.virtualenvs/nymms
```

Now you need to activate the virtual environment:

```
source ~/.virtualenvs/nymms/bin/activate
```

Now you can use either the instructions in *Using PIP* or *Installing From Source* above.

When you are finished using NYMMS you can deactivate your virtual environment with:

```
deactivate
```

Note: The deactivate command just unloads the virtualenv from that session. The virtualenv still exists in the location you created it and can be re-activated by running the activate command once more.

3.3.2 Permissions

NYMMS makes use of many of the Amazon Web Services. In order for the daemons to use these services they have to be given access to them. Since NYMMS is written in python, we make heavy use of the boto library. Because of that we fall back on boto's way of dealing with credentials.

If you are running NYMMS on an EC2 instance the preferred way to provide access is to use an instance profile. If that is not possible (you do not run on EC2, or you don't understand how to setup the instance profile, etc) then the next best way of providing the credentials is by createing an IAM user with only the permissions necessary to run NYMMS. You would then need to get that user's Access Key ID & Secret Key and provide them as the environment variables AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY.

Whichever method you choose, you'll need to provide the following permission document (for either the user, or the role):

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "ses:GetSendQuota",
        "ses:SendEmail"
      ],
      "Sid": "NymmsSESAccess",
      "Resource": [
        "*"
      ],
      "Effect": "Allow"
    },
    {
      "Action": [
        "sns:ConfirmSubscription",
        "sns:CreateTopic",
        "sns:DeleteTopic",
        "sns:GetTopicAttributes",
        "sns:ListSubscriptions",
        "sns:ListSubscriptionsByTopic",
        "sns:ListTopics",
        "sns:Publish",
        "sns:SetTopicAttributes",
        "sns:Subscribe",
        "sns:Unsubscribe"
      ],
      "Sid": "NymmsSNSAccess",
      "Resource": [
        "*"
      ],
      "Effect": "Allow"
    },
    {
      "Action": [
        "sqs:ChangeMessageVisibility",
        "sqs:CreateQueue",
        "sqs:DeleteMessage",
        "sqs:DeleteQueue",
        "sqs:GetQueueAttributes",
        "sqs:GetQueueUrl",
        "sqs:ListQueues",
        "sqs:ReceiveMessage",
        "sqs:SendMessage",
        "sqs:SetQueueAttributes"
      ],
      "Sid": "NymmsSQSAccess",
      "Resource": [
        "*",
      ],
      "Effect": "Allow"
    },
    {
      "Action": [
        "sdb:*"
      ],
      "Sid": "NymmsSDBAccess",
```

```
      "Resource": [
        "*"
      ],
      "Effect": "Allow"
    }
  ]
}
```

Note: If you want to provide even tighter permissions, you can limit the SNS, SDB and SQS stanzas to specific resources. You should provide the ARNs for each of the resources necessary.

3.3.3 Configuration

Please see the configuration page for information on how to configure `NYMMS`. Usually the configuration files are located in `/etc/nymms/config` but that is not a requirement and all of the daemons accept the `--config` argument to point them at a new config file.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search