# Notch Documentation

***Release 0.5***

**Andrew Fort**

April 10, 2013

# CONTENTS

Notch is system for building network management applications. Notch Agents expose a proxy interface to router and switch command-line interfaces (CLI) for client tool use, as well as managing device connections and authentication data.

See the Notch project page on Google Code for development information.

Contents:

CHAPTER

ONE

# AN INTRODUTION TO NOTCH

Notch exposes a programmatic interface to your network equipment, such as routers, switches, firewalls and load-balancers. Offering an HTTP based API and an Agent task which logs into devices, it can provide a consistent API for just any device that provides a command-line interface. Notch can be scripted using an available Python client library, and other language interfaces can be built using the Agent's JSON-RPC API.

Device usually ship with SNMP interfaces that work well for read-only data (e.g., counters), and fewer ship XML interfaces that can be used for configuration management. Command-line interfaces however, are still very commonly-used for configuration activities. Notch provides a common API for accessing the CLIs of a variety of router and switch operating systems, so you can use it to do anything you fancy, such as:

1. Configure network elements with templates based on any data source you can dream up.
2. Audit the turn-up of services you've configured.
3. Make large scale changes easier and results more certain.
4. Have a CLI for large (or small) networks talking to many routers at once.

Notch Agents tasks manage connections to device that clients have requested access to, keeping connections open until idle, to minimise login delays.

What you do with the connections made is up to you.

## 1.1 The problem

Tools like RANCID work very well for their intended purpose and have command execution decoupled from collection and parsing of router configurations, so they can be used to perform other ad-hoc or regular scripted activities. They are normally run on an operator's workstation or from a management server, often in one corner of the network. As a result, a number of problems exist:

1. Management stations talking to remote devices often incur significant network delay penalties. Networks with a wide geographic scope will always have some devices remote (latency-wise) from the management station.
2. Localised network partitions can severe the management station's path to the device, requiring the use of alternate or out-of-band access.
3. Because the low level transport (SSH, telnet) is used directly by the client program (e.g., *clogin*), there is no opportunity to compress the data if the transport does not explicitly offer this.
4. Running the same tool from multiple workstations generates many connections to the remote device (e.g., when several NOC staff attempt connections to the same device during an outage). Re-using a connection for operators with equal privilege minimises this problem.
5. Some device CPUs are inferior to those on a cheap workstation. SSH session setup may take many seconds. Running a few commands on a device and then disconnecting only to reconnect with a different client soon after is inefficient.

## 1.2 Why Notch?

Notch is designed to solve the above problems by offering a stateless API, by abstracting communications transport from any commands entered on them [3], by re-using existing connections where possible [4, 5] and by being a distributed system [1, 2].

## 1.3 Why the CLI?

Hacking the CLI in the way Notch unashamedly does is not an ideal solution for all network management problems. Its approach is not to support all requirements for all devices, but merely to give users what they are familiar with; the command-line interface.

Using the CLI has some disadvantages, such as hassles involved in screen-scraping and timing sensitive I/O.

Interestingly, there are hidden advantages to using the CLI. Vendors teach their CLI in training courses, and virtually every engineer interacting with a network device will have used it. With many more users, CLI bugs tend to get closed more rapidly and a stable interface already exists.

## 1.4 Components

Notch generalises the systems approach taken by RANCID, and splits any complete network management tool up into two components.

There is the Notch Agent (server), run on one or more computers that will make connections to your network equipment.

Human users and automated tasks employ Notch Client applications that instruct the Agents to perform work for them and return any results of that work.

The Notch Agent and Client communicate via HTTP using a JSON-RPC API.

For a basic installation, you install an Agent, the Notch Client library, and one or more Client applications.

In an advanced installation, one installs the Agent on multiple machines, install the Client library and applications on computers where they need to receive the results (such as their workstation, or a server used for periodic maintenance). The Client library can be configured to use specific Agents for particular network devices to geographically distribute the work on the devices. In addition, the client can load balance work between appropriate Agents to scale horizontally. This approach has been used to scale to a global network with tens of thousands of devices.

CHAPTER

TWO

# GETTING STARTED

## 2.1 System architecture

The Notch system is made up of two components. There is an Agent (server) process, run on one or more computers that will make connections to your network equipment. Human users and automated tasks employ client applications that instruct the Agents to perform work for them and return any results of that work.

For a basic installation, install an Agent, the Notch Client library, and one or more Client applications.

In an advanced installation, one installs the Agent on multiple machines, install the Client library and applications on computers where they need to receive the results (such as their workstation, or a server used for periodic maintenance). The Client library can be configured to use specific Agents for particular network devices to geographically distribute the work on the devices. In addition, the client can load balance work between appropriate Agents to scale horizontally. This approach has been used to scale to a global network with tens of thousands of devices.

## 2.2 Installation for new users

---

**Note:** If you have an existing Notch or Notch Client installation, you will need to perform some additional work to remove the old versions before proceeding with the following notes:

```
$ pip uninstall notch
$ pip uninstall notch.client
$ pip uninstall notch.agent
```

You should check your Python site-packages directories after this is completed to ensure you do not have any `notch` related files lying around.

---

First of all, Notch requires Python 2.6 or 2.7 (it is not expected to work with Python 3).

Notch is compatible with `virtualenv`, should you wish to install it into that. If so, switch into your desired virtual environment now.

If you do not use `virtualenv`, you will need to **run the following commands as root**.

Use `pip` to install Notch, if it's available on your system. If it's not, use `easy_install` to install `pip`:

```
$ easy_install pip
```

To install the Notch Agent (server which connects to routers, but has no client utilities):

```
$ pip install notch.agent
```

To install the Notch client library, used by tools like MrCLI to connect to the Notch Agent (and your network equipment):

```
$ pip install notch.client
```

Once you have completed this step, there is one final required library that has to be installed manually. Run this command:

```
$ pip install -e git+https://github.com/joshmarshall/tornadorpc.git@fda3e0e4#egg=tornadorpc-dev
```

## 2.3 Confirmation

You should now find the `notch-agent` binary installed in your PATH, and your Python interpreter should find the necessary modules:

```
$ python
Python 2.6.3 (r263:75183, Nov  6 2009, 15:46:51)
[GCC 4.2.1 (Apple Inc. build 5646) (dot 1)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import notch.agent
>>>
>>> # If you installed the notch.client package also,
>>> import notch.client
```

This merely checks that the files are installed correctly.

## 2.4 Next steps

Next, you'll want to configure your agent with information about your network devices and the authentication information used to log into them. See the *Notch Agent* page for more.

If you're an existing RANCID user, you'll want to see the *Quickstart for RANCID users* page.

CHAPTER

THREE

# QUICKSTART FOR RANCID USERS

If you already have RANCID installed, you should be able to get Notch up and running within 30 minutes of downloading the software by following this guide.

## 3.1 Configuration data

In the example, we'll keep our Notch configuration in `/usr/local/etc`.

First, find the directory that has directories with `router.db` files in them. For example:

```
/var/local/rancid/

Containing, for example, these ``router.db`` files::

  /var/local/rancid/access/router.db
  /var/local/rancid/backbone/router.db
  /var/local/rancid/mgmt/router.db
```

Use this `notch.yaml` master configuration file (given the above path):

```
device_sources:
  rancid_configs:
      provider: router.db
      root: /var/local/rancid/
      ignore_down_devices: True

options:
  credentials: /usr/local/etc/notch-credentials.yaml
```

Next, create the `/usr/local/etc/notch-credentials.yaml` file. This is site-specific, but here's an example of the types of things you can express:

```
- regexp: ^acc[0-9]+.*
  username: administrator
  password: notCisc0
  enable_password: l3ssLike7y
- regexp: .*
  username: administrator
  password: s4meOlth4nG
```

Devices like `acc1.bne` and `acc400.mel` will use the first credential, while all other devices will use the second credential.

---

**Note:** A `regexp: .*` wildcard record is not required. Your Agents will merely fail to connect to devices that don't match any other record. Configuring your Agents without wildcard authentication records is a recommended security precaution to avoid password leakage to other devices.

---

## 3.2 Converting the `.cloginrc` file

Since you use RANCID, you'll have a `.cloginrc` file you use for authentication credential data for RANCID. You can conver that (manually, at present) to a Notch credentials file, rather than creating one from scratch as in the above step.

While the cloginrc configuration file declares an option (a username, password or other option) for a glob per line, Notch places all the options for a group of devices (whose names match a regular expression) together in its configuration.

### 3.2.1 Example

cloginrc file:

```
add user ar* {automation}
add password ar* {/r0zza} {7p-Kz4PsLa01}
add user ?r* {cisco}
add password ?r* {foo} {bar}
add autoenable * 1  # See note
```

Equivalent Notch credentials file (e.g., `credentials.yaml`):

```
- regexp: ^ar.*
  username: automation
  password: "/r0zza"
  enable_password: 7p-Kz4PsLa01
- regexp: ^.r.*
  username: cisco
  password: foo
  enable_password: bar
```

---

**Note:** IOS devices (i.e., any using device module `dev_ios`) presently always `autoenable` in Notch. Allowing configuration of this and many other cloginrc type directives in the credentials file is coming in a near future release.

---

## 3.3 Running the agent

Start the `notch-agent` (a Tornado webserver running the Notch application, with a threadpool handling slow I/O) to answer requests:

```
$ notch-agent --config=/usr/local/etc/notch.yaml --logging=debug
```

That's it, you now have a Notch Agent running on `localhost:8080`!

You can even test the client from within the Python interactive interpreter:

```
$ python
Python 2.6.3 (r263:75183, Nov  6 2009, 15:46:51)
[GCC 4.2.1 (Apple Inc. build 5646) (dot 1)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import notch.client
>>> n = notch.client.Connection('localhost:8080')
>>> all_devices = n.devices_matching('^.*$')
>>> print len(all_devices)
2
>>> def cb(r):
...   print str(r)
...
```

```
>>> for dev in all_devices:
...   n.command(dev, 'show version and blame', callback=cb)
...
>>> n.wait_all()
<notch.client.client.Request object at 0x10193c510>
<notch.client.client.Request object at 0x1019091d0>
>>>
```

If you want a full-featured command-line front-end for Notch, please try MrCLI.

CHAPTER

FOUR

# NOTCH AGENT

The Notch Agent is the HTTP/JSON-RPC server at the core of the Notch system. Run one or more copies (aka *tasks*) of the Agent binary or WSGI application, referring to the same configuration data (the configuration and credentials data files). Keep in mind that some servers will run multiple processes of the application (e.g., `mod_wsgi`'s `processes=` directive), and this accounts to multiple tasks.

Clients refer to these agent tasks by the address/hostname and port of the Agent HTTP servers, like `notch.example.com:8443` or `notch.example.com:8080`. The client will look for the JSON-RPC endpoint at `/JSONRPC2`, but this is configurable in the Python client library if your environment requires.

The included Python client library can handle load balancing between multiple agent tasks as well as geographically distributed sharding. See the Notch Client source code for details on how to select different load-balancing strategies.

## 4.1 Application versions

The Notch Agent is distributed as a Python web-application. It is supplied with the Tornado web-server for non-production use, and as a Python WSGI application that can be deployed on any WSGI compatible web server. At this time it has been tested on Apache2 with `mod_wsgi` as well as `uWSGI`.

### 4.1.1 Stand alone

The standalone server (`notch-agent` should be in your `PATH`) can be used for testing. Start it with the `--config` argument pointing at your `notch.yaml` file, for example:

```
$ notch-agent --config=/usr/local/etc//notch.yaml --logging=debug
```

---

**Note:** The standalone server is not for production use. It does not execute tasks in parallel. It should be used for testing your initial configuration only before using a production server, such as Apache.

---

### 4.1.2 WSGI application

As WSGI servers use differing configuration methods to identify where the WSGI application code lies, the following data can be used to craft the necessary configuration for your server.

The module containing the WSGI application is:

```
notch.agent.wsgi.application
```

Which lives in the file (likely in your `site-packages` directory):

```
notch/agent/wsgi.py
```

## 4.2 Environment variables

The following environment variables are used to influence the Notch system at client or agent start-up time.

**Note:** The `NOTCH_CONFIG` variable must be set, pointing at the agent configuration file, prior to initialising the WSGI application. It has no effect when using `notch-agent`, which takes a `--config` argument, instead.

| Environment variable | Used by | Description |
|---|---|---|
| NOTCH_CONFIG | Agent | The path to the configuration file (default: None). |
| NOTCH_AGENTS | Client | A comma-separated list of agent host:port addresses. (default: None). |
| NOTCH_CONCURRENCY | Client | The maximum number of requests to keep in flight at any one time (default: 50). |

Client can programatically define which agents and concurrency they would prefer with the `agents=` and `max_concurrency=` arguments to the

## 4.3 Agent configuration

### 4.3.1 Master configuration

The Agent requires a configuration file, which is in YAML format.

There are two required top level sections, `device_sources` and `options`.

`options` contains the `credentials` attribute used to define the path to your credentials configuration file. In the `device_sources` section you can configure multiple device sources, which allow

### 4.3.2 Example

e.g., `/usr/local/etc/notch.yaml`:

```
device_sources:
    source1:
        provider: router.db
        root: /path/to/your/rancid_root_dir
        ignore_down_devices: True

options:
    credentials: /path/to/your/notch-config/credentials.yaml
```

The `provider` attribute has two currently accepted values:

> `router.db`: Loads device information (device names and vendor module type) from RANCID router.db configuration files. The agent asynchronously refreshes its data every few minutes.
>
> `dnstxt`: Use DNS A queries to find IP address information, and DNS TXT queries to retrieve `v=notch1` prefixed records for the purpose of determining vendor module type information for the device, e.g.,

```
ar1.foo.int.example.com. IN A   10.0.22.75
ar1.foo.int.example.com. IN TXT "v=notch1 device_type=juniper"
```

#### Credentials file

User credentials, that is the usernames, passwords or SSH keys used when connecting to network devices, are configured in the credentials configuration file.

If you haven't already, you should now go and create Notch specific users (e.g., `automation`) on your administrative TACACS+ or RADIUS server. Change the passwords on the server and in your credentials configuration file on a regular basis.

---

**Note:** Only a limited range of system administrators need know these passwords. Make sure you set the permissions on your password file appropriately:

```
$ chown notchuser /opt/local/etc/notch.yaml
$ chmod 700 /opt/local/etc/notch.yaml
```

---

### 4.3.3 Credential Attributes

The credentials file is a YAML repeated block, consisting of attributes named `regexp`, `username`, `password`, `enable_password` and `ssh_private_key`.

`regexp` is a string regular expression. Device names matching this regular expression will be use this credential. For each request, the filter is evaluated in Last Match mode. Start with any rules that match an individual device, followed by those which match by less restrictive regular expressions. If you require one, place any `regexp: .*` defaults at the end of the configuration file.

`username` and `password` should be understood, `enable_password` is the "enable" password often used on Cisco or other platforms supporting TACACS+. `ssh_private_key` is an ASCII-armored form of the SSH private key data used for matching devices.

#### Example credentials file

In the example below, the border routers (e.g., `br01.bne03`, `br1.mel07`) will use the `automation` username with the `tBRpass` and the predictable enable password. Every other device will use the `ssh_private_key`, whilst stil using the `automation` username.

`credentials.yaml`:

```
-
  regexp: ^br[0-9].*
  username: automation
  password: tBRpass
  enable: c15c0
-
  regexp: .*
  username: automation
  ssh_private_key: "-----BEGIN RSA PRIVATE KEY-----\n..."
```

There is *no need* for a trailing – (it adds an empty block which is ignored by the parser).

CHAPTER

FIVE

# NOTCH.CLIENT

CHAPTER

SIX

# `NOTCH.CLIENT.LB_TRANSPORT` – RPC LOAD BALANCER

CHAPTER

SEVEN

# CHANGELOG

## 7.1 Version 0.4.3 (Jan 20 2011)

- **New device support.**
  - Alcatel OmniSwitch (tested on 6855): `omniswitch`
  - Nortel ESR: `nortel_esr`
  - Nortel ESU: `nortel_esu`
  - Nortel/Bay BaySwitch: `nortel_bay`
- Harmonise API for telnet, expect+Paramiko transports.

## 7.2 Version 0.4 (Jul 22 2010)

- **New device support.**
  - Arbor TMS/CP: `arbor`
  - Juniper Netscreen: `netscreen`
- **Provide apache2 `mod_wsgi` configuration file example**
  - `mod_wsgi` deployment is recommended and well tested.
- `command()` responses are always base64 encoded by the Agent.
- `get_config()` implemented for Paramiko devices (e.g., `junos`).
- `download_file` implemented for Adva FSP and Paramiko devices.

## 7.3 Version 0.3 (May 27 2010)

- **New device support.**
  - Alcatel 7750/7450/7210/7705 (TimOS): `timetra`
  - Movaz/Adva FSP: `adva_fsp`
  - DASAN/Siemens PON/ONU (running NOS): `nos`
  - BATM/Telco/Temarc switches (running BiNOS): `telco`
- New abstract transport for building device modules based on SSHv2 and either exec-sessions or expect'ed login-sessions.
- Devices can support multiple connection methods.

- Devices will retry in cases like EOF errors.
- New `devices_info` API method, similar to `devices_matching`, that returns device metadata.
- Support SSHv2 servers on IOS and Alcatel/Timetra devices.
- Credential records can set the connection method and auto-enable property.

## 7.4 Version 0.2 (April 3 2010)

- Initial Cisco IOS device support (telnet only).
- Connection (client) class can `kill_all()` outstanding requests.
- Client can now query Agent's device inventory (via new `devices_matching` API method) using a regular expression.
- Agent error reporting improved.
- Device modules now live in the notch/agent/devices/ directory.
- Devices are disconnected when idle (after `Device.TIMEOUT_IDLE`).
- Mr. CLI updated significantly and split off into separate package, see http://code.google.com/p/mr-cli

## 7.5 Version 0.1 (March 25 2010; initial release)

- Initial release.
- JunOS device support (via SSH2 only).

CHAPTER

EIGHT

# INDICES AND TABLES

- *genindex*
- *modindex*
- *search*