Headless Network-based Configuration Framework with Ansible and Flask

Pablo Collado Soto

CDT Technical Report 2021-6

Contents

Listings

1 Introduction

Configuration management refers to the field concerned with handling the configuration of a collection of devices. By configuration we are referring to aspects such as the services a given system is running as well as the settings used to run said service.

As the number of elements in a system grows, configuration management becomes increasingly complex. We then need to find a way of tackling this complexity so that the time it takes to get a system up and running is as short as possible. We also need to address service faults and the reconfiguration of nodes in a timely manner. Whilst all of these procedures may be manually taken care of by an administrator, doing so is a rather error prone process with a very low work throughput.

With computing offering facilities that are applicable to more fields with each passing day, we have seen cheap, headless systems such as the *Raspberry Pi* single board computer rise in popularity. These systems are capable of running a full-fledged `linux`-based system, which means they can leverage all the existing tools and systems targeting that platform. These boards are used at the *Yebes Astronomical Observatory* for controlling several modules in the reception chains for radio-telescopes thanks to in-house software. The number of boards has been steadily growing, which puts a burden on those in charge if configuring new additions. Our system tries to automate the deployment of new controllers in terms of software without the need of any human intervention whatsoever.

What's more, the *Yebes Astronomical Observatory* develops technology for other sites. This system can ease the effort needed by external staff to configure new equipment, thus making *Yebes's* technology a bit more attractive.

In order to accomplish its goal, our system orchestrates automation tools such as `Ansible` with auxiliary services. These technologies work in consonance in an effort to provide a framework that's capable of automating the deployment and provisioning of new systems at the *Yebes Astronomical Observatory*.

2 Definitions

In order to ensure the following discussion is easy to follow we'll define several terms that might not be well known by the reader:

- **`Daemon`**: Program running on a machine that's not intended to process user input directly. `Daemons` are the implementation of services as pervasive as `HTTP Servers` or `SSH Servers` for instance. They will usually read a configuration file when starting so that they make the predefined configuration effective and dump output to a *log file* we can later consult. These daemons are commonly managed by the *system daemon* (`systemd`) on `linux`-based systems.

3 Getting the code

Even though we have included most of the framework's components as listings on the document, one can browse the entire codebase (which contains additional documentation) and gain access to any desired resources at the RaspberryPi Provisioning Group on the `git.oan.es` server.

4 System components

As stated in the report's introduction, our system is composed by several independent software services. We'll now describe each of them to provide the background needed to tackle the rest of the document.

4.1 Ansible

`Ansible` is the **backbone** of our system. It is a *configuration automation tool* that'll let us configure new and existing systems over a network connection. `Ansible` is based on the classic *client-server* architecture. We differentiate between a *controller* and one or several *targets*. Their roles are:

- ***Controller***: The machine acting as the controller will configure one or several targets over a network connection. Ansible's default workflow is having the controller initiate the configuration of the targets, that is, the controller is the one bootstrapping the configuration.
- ***Target***: These machines are the ones we'll apply a given configuration on. The needn't have any dependencies installed for ansible to act on them. This alleviates a lot of preliminary efforts and enables a truly headless configuration.

The configurations `ansible` is to apply on targets are defined by means of a *playbook*. A *playbook* is a simple text file with a `YAML` syntax that defines the `tasks` the controller is to run against the targets. These tasks can be grouped into *roles* and each of these *roles* can run against different sets of targets too.

The following snippet portrays a simple *playbook* along with several caveats:

```
---
    # The targets to run the playbook against.
    # This MUST be left as is for headless provisioning
- hosts: all

    # The user that'll make the changes in the remote system.
    # As we are targeting RPis this will usually be 'pi'
    remote_user: pi

    # Method to use when privilege escalation is needed for a
    # given task
    become_method: sudo

```

```
14      # These variables can be use elsewhere in the document by including
15      # their name between double curly braces ({{}}). Note both
16      # 'services' and 'dependencies' are lists instead of simple
17      # variables.
18      vars:
19          # URL to pull the deployment files from.
20          repo_url: "http://git.oan.es/rpi-provisioning/test.git"
21
22          # Path where the files will be deployed.
23          repo_path: "/home/pi/deployment"
24
25          # Services we are to enable.
26          services:
27          - pcalcontrol.service
28          - peltiercontrol.service
29
30          # Packages we need for the deployment of services and
31          # for the deployed software.
32          dependencies:
33          - git
34          - python-netifaces
35          - python-serial
36
37      # List of tasks to run against the chosen target
38      tasks:
39      # Note '{{}}' need to be surrounded by quotes when they are
40          # the first character in a line.
41      - name: Update the APT repositories and install dependencies
42          apt:
43          name: "{{ dependencies }}"
44          update_cache: yes
45          become: yes
46
47      - name: Pull the current repository with the latest changes
48          git:
49          repo: "{{ repo_url }}"
50          dest: "{{ repo_path }}"
51          depth: '1'
52          version: main
53
54      # This task leverages a loop to run for several target services.
55          # The loop parameter will iterate over the 'services' variable
56          # we defined before and on each iteration, 'item' will evaluate
57          # to the current service file.
58          # The become parameter allows this task to be run with privileges,
59          # something that's needed for installing new services.
60      - name: Link the unit files so that systemd picks them up
61          file:
62          src: "{{ repo_path }}/{{ item }}"
63          dest: /etc/systemd/system/{{ item }}
64          owner: root
65          group: root
66          state: link
```

```
67          mode: "0644"
68          follow: no
69          loop: "{{ services }}"
70          become: yes
71
72      - name: Make systemd pick up the new unit files
73          systemd:
74          daemon_reload: yes
75          become: yes
76
77      - name: Enable and start the services
78          systemd:
79          name: "{{ item }}"
80          state: started
81          enabled: yes
82          loop: "{{ services }}"
83          become: yes
84
85      - name: Disable the provisioning request service
86          systemd:
87          name: request_provisioning.service
88          state: stopped
89          enabled: no
90          become: yes
```

Listing 1: A sample playbook.

The example we have provided on listing 1 is rather simplistic when compared with all the functionality provided by *playbooks*. However, given the configurations engineers are usually to deploy, it adapts quite well to the needs of the staff and should accommodate most use cases with minor tweaks.

`Ansible` will leverage `SSH` to log into targets and configure them. That's why the best approach to design a truly headless system is to add the server's *public SSH key* to each of the target's *authorized keys* file under `/.ssh/authorized_keys`. That way we'll avoid the server depending on us accepting a connection to a new unknown machine: everything will instead work seamlessly. This key is added to targets through an image modification process as we'll see on a later section treating that particular topic.

If there is one thing to love about `ansible` is its magnificent documentation which can be found here. One can query all the available modules on this portal as well as the parameters they accept and how they behave. One can also look into more advanced functionality offers regarding privilege escalation and *playbook* testing, among others. We do want to warn the reader about how "strict" the `YAML` syntax can be. We encourage anybody writing a *playbook* to be extra vigilant with the indentation and that they use spaces instead of tab characters (`t`). We also recommend using a text editor capable of rendering the document's white space so that one can verify to be respecting the format. The error messages regarding format are rather cryptic and hard to debug, so it's better to try and avoid them altogether.

We'll revisit some of `ansible`'s capabilities later on when we delve deeper into our system's architecture.

4.1.1 Installation on Debian platforms

Installation on *debian* based systems is just a matter of running `sudo apt update && sudo apt -y install ansible`. This will provide us with the `ansible-playbook` command, which is the one we'll use to trigger a playbook against a target automatically.

4.2 Flask

`Flask` advertises itself as a *micro web framework* that enables a user to write web applications using the *python* language. We'll be using `flask` to implement an `HTTP` server that'll listen for requests and trigger an `ansible` *playbook* against the client who made the request.

We find then that `flask` will play a small but crucial role in the overall system. We will be using it in a somewhat "unorthodox" way in the sense that we won't be providing a regular `HTTP` server offering web pages to users: we'll instead be implementing an `HTTP` interface that configuration targets can employ to trigger their own configuration.

`Flask`'s documentation page can be found here. It's got way more information than what we really need to know, but it could prove to be useful should the provisioning server need to be augmented in the future.

4.2.1 Installation on Debian platforms

Just like with `ansible` installing `flask` on a system-wide level can be accomplished with `sudo apt update && sudo apt -y install python3-flask`. We have decided to instead run `flask` on a *python3* virtual environment so that the `flask` installation is isolated from the rest of the system. We carry our the entire procedure automatically through the `launch_provisioning_server.sh` script which we include on listing 2.

```
#!/bin/bash

# If the virtual environment doesn't exist create it
    # and install flask right away. Otherwise
    # source the existing virtual environment.
if [ ! -d fenv ]
then
    echo "Creating the fenv virtual environment"
    python3 -m venv fenv

    echo "Activating the fenv virtual environment"
    source fenv/bin/activate

    echo "Installing flask..."
    python3 -m pip install flask
else
    echo "Activating the fenv virtual environment"
    source fenv/bin/activate
fi

# We tell flask we are running in a development environment
    # so it doesn't complain!
```

```
23 echo "Running the provisioning server"
24 FLASK_ENV=development python3 ./provisioning_server.py
```

Listing 2: Automatic server activation.

4.2.2 Running the server as a daemon

We have also written a `unit file` letting the `flask` server run as a *daemon* controlled by `systemd`. We are including it on listing 3. It currently runs the provisioning server within a virtual environment, but could easily be adapted to run it together with a system-wide installation. Installing the file is just a matter of running `sudo cp <flask_unit_file> /etc/systemd/system` on the machine acting as the provisioning server. After that, one should be capable of seeing the service through `systemctl status <unit_file_name>`. Us usual, we can enable it at boot with `sudo systemctl enable <unit_file_name>` and start it with `sudo systemctl start <unit_file_name>`. This is by no means a guide on using `systemd`: take these commands with a grain of salt and adapt them to your system.

```
1 [Unit]
2 Description=Flask-based Provisioning Server
3 After=network.target
4
5 [Service]
6 User=server-username
7 WorkingDirectory=/path/to/provisioning_server/
8 ExecStart=/bin/bash /path/to/provisioning_server/launch_provisioning_server
      .sh
9 Restart=always
10
11 [Install]
12 WantedBy=multi-user.target
```

Listing 3: Provisioning server unit file.

4.3 Raspberry Pi image modifier

Given the provisioning of the system we are to configure needs to be headless, we need to make this target initiate the entire process. This lets us know crucial aspects such as its `IP` address so that we know who to aim the *playbook* at for instance. Given the provisioning server is implemented as an `HTTP` server implemented with `flask`, the newly booted machine will interact with it through the `HTTP` protocol.

We then need to somehow make the *raspberry* perform an `HTTP GET` request against a particular `URL` within the provisioning server. We can do so by means of tools like `curl`, which is present on most `linux`-based distributions by default. The `Raspberry Pi OS` is among these distributions, so leveraging `curl` doesn't imply handling a new dependency.

Even though `curl` is shipped by default, we still need to "tell" the *raspberry* what `URL` to request and we need it to perform such action on boot. We can actually pull this off by modifying the *operating system image* (the `*.img file`) we burn onto the `SD` card that acts

as the *raspberry*'s main storage unit. `Linux` offers the possibility of mounting the image on a *loopback* device that allows us to modify said image directly on an arbitrary *OS*. We just need to provide a *mountpoint* (i.e. a *directory*) to mount the image on and it'll be offered just like any other regular directory.

The catch with the above strategy is that we need to account for an *offset* when mounting the image, as it contains all the boot sectors and whatnot before the actual data partitions. Thankfully, we can use the `fdisk` utility to gather all the information we might need. Once the image is mounted we'll install a *unit file* just like we did with the `flask` server before that'll trigger an `HTTP` request on boot. This service will be disabled by the configuration through the *playbook* so as to avoid reconfiguring the target at hand on every boot.

The key idea here is that the `URL` we specify on the *unit file* we are adding to the image controls the deployment we are to install on the target itself. This means adding an entire new deployment is just a matter of adding a *server endpoint* (more on that later) and changing the `URL` on the service we install on a given system.

We'll also leverage this image modification process to add the provisioning server's *public SSH key* to the list of authorized keys for this system. That way, the server won't need any additional arguments to run the playbook or depend on user input of any kind. We'll also "headlessly" enable `SSH` on the target by `touch`ing (i.e. creating an empty file) with name `SSH` on the `/boot` partition. The *raspberry* checks it at boot time to decide whether to enable the `OpenSSH` server automatically or not.

You can find a `bash` script automating this image modification on listing 4 and a sample *unit file* automating the request on listing 5.

```
1  #!/bin/bash
2
3  # General paths and variables. Remember to change these if you move
       auxiliary files around!
4  provisioning_service="request_provisioning.service"
5  services_path="/etc/systemd/system"
6  enabled_services="/etc/systemd/system/multi-user.target.wants"
7  pub_key="ansible_controller_key.pub"
8
9  # Check we are running as root!
10 if [ $EUID -ne 0 ]
11 then
12     echo "Run me as root please :P"
13     exit -1
14 fi
15
16 # Check we only received one argument!
17 if [ $# -ne 2 ]
18 then
19     printf "Usage: %s <image_file> <mountpoint>\n" $0
20     exit -1
21 fi
22
23 # Check if the file exists
24 if [ ! -e $1 ]
25 then
```

```
26      echo "Couldn't find the image at $1. Quitting..."
27      exit -1
28  fi
29
30  # Check the mountpoint exists
31  if [ ! -d $2 ]
32  then
33      echo "Couldn't find the mountpoint at $2. Quitting..."
34      exit -1
35  fi
36
37  printf "Analyzing image on file %s\n" $1
38
39  # Get the sector size and partition offsets by parsing the output with awk
40  sector_size=$(fdisk -l $1 | head -n 2 | tail -n 1 | awk '{print $8}')
41  boot_offset=$(fdisk -l $1 | tail -n 2 | head -n 1 | awk '{print $2}')
42  rootfs_offset=$(fdisk -l $1 | tail -n 1 | awk '{print $2}')
43
44  printf "Detected sector data:\n\tSector Size   -> %8d bytes\n\tBoot Offset
        -> %8d sectors\n\tRootfs Offset -> %8d sectors\n"\
45          $sector_size $boot_offset $rootfs_offset
46
47  # Mount the boot partition. It's a FAT32 (vfat) filesystem and we are
      mounting it on a "loop" device
48  # Note the $(()) is what bash calls an arithmetic expansion and let's us do
       "math" within it.
49      # You can read more on that on bash's manpage.
50  printf "Mounting the boot partition...\n"
51  mount -t vfat -o loop,offset=$(($sector_size * $boot_offset)) $1 $2
52
53  # Enable SSH headlessly
54  printf "Touching the ssh file to headlessly enable SSH...\n"
55  touch $2/ssh
56
57  # Sync the changes and unmount the boot partition. Sleep a couple of
      seconds so as not to rush stuff.
58      # We did run into some issues without this small delay...
59  printf "Syncing changes and unmounting the boot partition...\n"
60  sleep 2
61  sync && sudo umount $2
62
63  # Mount the rootfs partition. Its an EXT4 filesystem and we are mounting it
       on a "loop" device
64  printf "Mounting the rootfs partition...\n"
65  mount -t ext4 -o loop,offset=$(($sector_size * $rootfs_offset)) $1 $2
66
67  # Copy the provisioning service
68  printf "Copying the $provisioning_service file to $services_path\n"
69  cp $provisioning_service $2$services_path
70
71  # Enable the service at boot
72  printf "Enabling $provisioning_service through a link on
      $2$enabled_services/$provisioning_service\n"
```

```
ln -s "$services_path/$provisioning_service" "$2$enabled_services/$provisioning_service"

# Move the public key to the /home/pi/ssh directory so that we can then use RSA-based authentication
printf "Moving public SSH key into /home/pi/.ssh/authorized_keys"
mkdir "$2/home/pi/.ssh"
cp $pub_key "$2/home/pi/.ssh/authorized_keys"

# Sync the changes and unmount the boot partition. Sleep a couple of seconds so as not to rush stuff.
    # We did run into some issues without this small delay...
printf "Syncing changes and unmounting the rootfs partition...\n"
sleep 2
sync && sudo umount $2

printf "Finished modifying the image! Ready to burn :P\n"

exit 0
```

Listing 4: Automatic modification of the OS image.

```
[Unit]
Description=Request Ansible Provisioning
After=multi-user.target network.target
Requires=network-online.target

[Service]
Type=simple
ExecStart=/usr/bin/curl http://<provisioning_server_ip>:<provisioning_server_port>/<provisioning_configuration>
Restart=on-abort

[Install]
WantedBy=multi-user.target
```

Listing 5: Sample confgiuration trigger service.

Even though the process has been automated, the entire procedure can be manually replicated by running the commands found on listing 4 one by one and adding the desired modifications. One can also modify the script itself. We have just provided the most common steps as a convenience but they are by no means compulsory. One could even burn a *vanilla* (i.e. unmodified) *Raspberry Pi OS* image onto a board and then manually install the service file or just explicitly running a *playbook* against it explicitly, scrapping the system altogether. We want to make clear that this system offers "a way" of doing things without imposing anything on the end user.

5 System logic and flow

Knowing the components we are to employ to offer an automatic configuration framework we can then look into how a new system goes from booting to being automatically configured.

We'll look into each of the steps in the process.

5.1 Booting

A newly booted *raspberry* will be first assigned an `IP` address by a `DHCP` server. Another approach would be assigning it a static address, but that would imply it's more adequate to run `ansible` against the *raspberry*, given we know it's address beforehand. That's why we'll assume we have a `DHCP` server in play, the automatic headless provisioning loses a lot of its appeal otherwise.

5.2 Triggering the configuration

Once the *raspberry* boots, `systemd` will eventually run the *provisioning service* we added by modifying the image we burnt onto the `SD` card. This will trigger an `HTTP` request on the *provisioning server*.

5.3 Server request processing

The *provisioning server* will obtain then execute the function associated to the request's `URL`. This function will obtain the target's `IP` address from the request itself (let's not forget the source `IP` is carried on the `IPv4` datagram where the `HTTP` request is encapsulated). It'll then run the *playbook* associated to the `URL` against the `IP` that made the request.

5.4 Cleanup

After all the pertinent tasks are carried out on the target, the service triggering the process will be disabled so that it doesn't start at boot. This will avoid having the target reconfigure itself (possibly loosing data files and whatnot) on each boot.

We then find that the logic driving the system itself is not complex nor long and tedious. We'll now look into how we can augment the server to account for more deployment options and how to make more *playbooks* available.

6 Server endpoints: adding new configurations

The structure of our `flask`-based server is rather simple. Listing 6 contains the server's code. `Line 16` defines a *demonstration endpoint*. The function defined on `line 17` will be run whenever an `HTTP` request is made to the `http://<server_ip>/demo_endpoint`. Then, adding a new deployment is a matter of copying and pasting `lines 16 - 44` and modifying the *endpoint*'s name together with the path to the target *playbook*, which, in our case, is defined on `line 28`.

Note how the path to the *playbook* is relative to the directory to where the server file is located (`.`). This is important, these *playbooks* reside on the **server**. One will find a `playbooks/` directory there under which the *playbooks* one wants to make available should

be located. One might think that, as along as the path to a *playbook* is correct and the user running the server can read it it'll work. That's actually correct, but we strongly encourage storing all the playbooks in a centralized directory. We have decided to deploy this server through the `git` version control system, that is, we'll update the server's contents by running `git pull` within the directory containing all the files. If *playbooks* aren't located under this root server directory, they won't be a part of version control and will have to be manually updated. This is bound to provoke some errors in the long run...

```
import flask, subprocess, pathlib, os

# Variables controlling the server's behaviour
USER_NAME = 'pi'
SSH_KEY = 'ansible_controller_key'

# Choose whether to print messages to STDOUT (DBG is True) or to a log
    # file on the same directory (DBF is False)
DBG = False

# Instantiate a Flask app
app = flask.Flask(__name__)

# Define an endpoint. The 'demo_endpoint' text is the one YOU SHOULD
    INCLUDE in the
    # request_provisioning.service file when modifying a Raspberry Pi OS
    image!
@app.route("/demo_endpoint")
def trigger_demo_provisioning():
    # Print the IP address of the client that made the request
    print("Provisioning client @ {}".format(flask.request.remote_addr))
    if not DBG:
        # Run ansible-playbook against the IP of the client that requested
    provisioning
            # Note the last element of the list is what controls the
    configuration that
            # will be deployed on any given RPi!
        out = subprocess.run(['ansible-playbook',
                        '-i', "{},".format(flask.request.remote_addr),
                        '-u', USER_NAME,
                        '--private-key={}'.format(SSH_KEY),
                        './playbooks/demo_playbook.yml'],
                        capture_output = True)

        # If not debugging, print ansible's output to a log file on this
    directory
        pathlib.Path("ansible.log").write_text(out.stdout.decode())

        # Return an answer based on anisble-playbook's return code!
        return flask.jsonify({'provisioning_ok': out.returncode == 0}), 200
    else:
        # If debugging print the ansible-playbook's output to STDOUT (i.e.
    the running terminal or
            # the service's log, which we can querry with journalctl)
```

```
39          out = os.system('ansible-playbook -i {}, -u {} --private-key={} {}'
    .format(
40                  flask.request.remote_addr, USER_NAME, SSH_KEY, './playbooks/
    demo_playbook.yml'
41          ))
42
43          # Return an answer based on anisble-playbook's return code!
44          return flask.jsonify({'provisioning_ok': out == 0}), 200
45
46 # Define a default endpoint for checking whether the server is listening as
       it should
47 @app.route('/')
48 def check_server():
49      print("Got a request from {}!".format(flask.request.remote_addr))
50      return flask.jsonify("You see me! :P"), 200
51
52 # Run the flask application
53 if __name__ == '__main__':
54      # Bind to (i.e listen for connections on) every network interface.
55          # This can be changed to only allow hosts on the LAN to
56          # trigger configurations.
57      app.run(host = '0.0.0.0', port = 8080)
```

Listing 6: The provisioning server.

7 Testing new playbooks

Even though *playbooks* will be run when booting a new system, triggering test runs in this way is rather unfeasible. That's why we would like to point out that one may run these *playboooks* explicitly from any system that's got `ansible` installed. Doing so is just a matter of running the command found on listing 7. One need only provide the target's `IP` address and have a local copy of the *playbook* to test.

```
1 # Command breakdown:
2      # ansible-playbook: Program executing a playbook against a target.
3      # Options:
4          # -k: Ask for the SSH login password. That way we don't have to
    configure keys.
5          # -u <target_name>: Username to run the tasks as on the target.
6          # -i <target_ip>,: The target's IP. The trailing comma IS NEEDED,
    as it
7              # tells ansible-playbook this is a list of targets and not a
    targets
8              # filename.
9          # <path_to_playbook>: Path to the playbook to run.
10
11      # One can optionally use the -K option to provide the password to use
    for
12          # privilege escalation.
13
```

```
14     # As always, the commands manpage (accessible with man ansible-playbook
       )
15         # sheds a ton of light on its operation.
16 ansible-playbook -k -u <target_username> -i <target_ip>, <path_to_playbook>
```

Listing 7: Explicitly running a playbook.

8 Closing thoughts

This report describes the anatomy of a system providing headless configuration of new machines together with the way of adding new configurations to it, thus making it "infinitely extensible". We hope this small framework can take some work off the staff at the observatory whilst making large-scale configuration changes easier. We also believe this could help staff configure equipments sent to other observatories in an easier way by providing this system together with a working *playbook* to whoever might need it.