paradrop Documentation

Release 0.0.1

Paradrop Labs

November 30, 2016

Contents

This section of the documentation is devoted to describing the apps that run on Paradrop.

If this is your first time seeing Paradrop, please start with the Getting Started page.

Contents:

CHAPTER 1

Getting Started

This will quickly take you through the process of bringing up a Hello World chute in a virtual machine on your computer.

NOTE: As of release 0.1, pdbuild is built around using Ubuntu. We will eliminate this requirement soon, work arounds can be found at *What if I don't have Ubuntu?*.

1.1 Environment setup

0. Prerequisites:
   - Packages: Python 2.7, python-pip, python-dev, libffi-dev, libssl-dev
   - PyPI: pex
   - When you install build tools you may have to run: `sudo pip install pypubsub --allow-external pypubsub`
1. Install our build tools (`pip install pdtools`).
2. Clone our instance tools.
3. Setup instance tools `pdbuild.sh setup`
4. Boot local testing VM `pdbuild.sh up`
5. Install instance dependencies `pdbuild.sh install_deps`
6. Build the tools to go into testing VM `pdbuild.sh build`
7. Push tools into VM `pdbuild.sh install` (NOTE: sometimes this fails, please check Known Issues)
8. Check the installation `pdbuild.sh check`

1.2 Installing chutes

First you must register an account from our CLI: `paradrop register`. This will setup a private key on your computer which allows you to access our platform.

- Clone the Paradrop example apps.

Install **hello-world** chute:

```
cd <apps-repo>/hello-world
paradrop chute install localhost 9999 ./config.yaml

Result:
...
Chute hello-world create success
```

As a simple use case, **hello-world** starts an nginx server in the chute. To access this, visit `localhost:9000` in any web browser, you should see:

```
Hello World from Paradrop!
```

Running `paradrop chute stop localhost 9999 hello-world` will stop the chute, if you refresh the webpage, you should no longer see the Hello World message.

CHAPTER 2

Paradrop Chute Tutorials

This page details out information about advanced chute architecture and installation. We assume you have already gone through Getting Started.

2.1 Testing on your development computer

To keep the development process for Paradrop as simple as possible, we heavily encourage and support developers testing their chutes on a virtual machine (VM).

2.1.1 Wi-Fi in virtual machines

In order to support development on a virtual machine, you most likely need a Wi-Fi device (otherwise it wouldn't be a router would it??). The instructions below will show how to enable Wi-Fi specifically for USB adapters, but other internal Wi-Fi cards should follow similar steps.

Plug in the WiFi card, on Ubuntu, run `lsusb`, you should see:

```
Bus 002 Device 005: ID 148f:5372 Ralink Technology, Corp. RT5372 Wireless Adapter
```

Make note of the *Bus* and *Device* numbers, in this case 2 and 5.

When you go to launch your VM with a Wi-Fi device, simply run the command:

```
sudo pdbuild.sh up wifi-2-5
```

You need `sudo` access because the VM needs to pull in the USB device, which is privileged.

You can verify that the WiFi adapter is inside of the VM by running:

```
pdbuild connect

(amd64)ubuntu@localhost:~$ iw dev
phy#0
    Interface wlan0
        ifindex 4
        wdev 0x1
        addr 7c:dd:90:8f:c2:5e
        type managed
```

This will SSH you into the VM and print out information about the WiFi adapter, if this print out is blank, also try `iw phy` which prints out physical information about the Wi-Fi radio.

2.2 Wi-Fi Enabled Chutes

Here we will describe how to install a chute that utilizes a WiFi radio in the router.

2.3 Chute: Virtual Router

The chute we will install is called *virtual router* it is as simple as it sounds. This chute will setup a fully functional virtual router inside of the real router hardware (or VM). This is useful to demonstrate the full capability of Paradrop, which will setup and establish the chute, and tie together the networking components needed to allow the chute to function as a router.

Setup:

- Make sure your VM is alive and has WiFi (explained above).
- Make sure you are logged in or registered using pdtools.

Install the chute:

```
cd <example_apps>/virtual-router
vim config.yaml
#Setup ssid and password (defaults to "Paradrop-Network" and "ParadropRocks!")
paradrop chute install localhost 9999 config.yaml

... (install output)
Chute virtual-router create success
```

Now use your laptop or phone and search for the SSID you created, you should be able to associate to it and use it normally. You can verify you are using the chute for internet by stopping it:

```
paradrop chute stop localhost 9999 virtual-router
Stopping chute...

Chute virtual-router stop success
```

CHAPTER 3

Chute Configuration Files

There are 2 files that are important to the use of Paradrop and installing chutes.

- *config.yaml* - contains high level information about the chute for the host OS (like WiFi SSID's that need to be setup)
- *Dockerfile* - contains internal chute actions to setup the chute (like what OS version to use)

Eventually these will all fold into one glorious configuration file, but at this early stage we keep them separate.

3.1 config.yaml

Check here for information specific to the config.yaml file and its options.

3.2 Dockerfile

Check here for information on the Dockerfile and how it works.

CHAPTER 4

Installing Paradrop on hardware

Paradrop is distributed as a "snap" or an application that runs on Snappy Ubuntu. You can run snappy on any x86 or armv7 board (Raspberry Pi Gen 2 or Beagleboard Black supported!)

To setup Paradrop you need to install snappy on your hardware of choice and then have snappy install paradrop. This is a temporary method until more robust installation tools are finished.

First flash the board with the snappy image, see `flashing`.

Next install docker:

```
ssh into the router
sudo snappy install docker
```

From your development machine (because you cannot install unauthorized snaps internally, only using `snappy-remote` for now).

Next install a few required programs not in the Snappy package system yet:

```
wget https://paradrop.io/storage/snaps/dnsmasq_2.74_all.snap
snappy-remote --url=ssh://<ip>:8022 install dnsmasq*.snap

wget https://paradrop.io/storage/snaps/hostapd_2.4_all.snap
snappy-remote --url=ssh://<ip>:8022 install hostapd*.snap
```

Finally, install Paradrop, unfortunately this is not an officially supported Snappy package yet so it must be installed manually using snappy tools:

```
#From the Paradrop github repo:
cd paradrop
python setup.py bdist_egg -d ../buildenv
cd ..
[ ! -f snap/bin/pipework ] && wget https://raw.githubusercontent.com/jpetazzo/pipework/3bccb3adefe81b
chmod 755 snap/bin/pipework

rm -f snap/bin/pd

pex --disable-cache paradrop -o snap/bin/pd -m paradrop:main -f buildenv/
rm -rf *.egg-info

snappy build snap
snappy-remote --url=ssh://localhost:8022 install <snap-location>
```

CHAPTER 5

Provisioning Devices

Once you've got paradrop up and running on hardware or on a virtual machine you'll need to *provision* the software. When a brand new router starts for the first time, it doesn't have a place in the world yet. It doesn't even know its name! Additionally, the provisioning process secures your software to you and only you– its an important security step.

The steps listed here are an intermediate process. Provisioning will occur during the installation process, check back soon for updates.

5.1 Provisioning Routers v0.1

Before you begin, make sure you have an installed version of the CLI tools and an account with paradrop. You will need to be logged in for every instruction that follows:

```
pip install pdtools
paradrop register
```

or if you already have pdtools installed:

```
paradrop login
```

Please choose usernames and passwords that are at least 8 characters.

Create a new router with the server. All of your routers have to have unique names, but lets use aardvark:

```
paradrop router-create aardvark
```

Once the creation process is finished see all of your owned chutes and routers with:

```
paradrop list
```

If this is your first time, You'll only see your single new router as part of its `pdid` (Link forthcoming.) This is the id of your router to the rest of the world:

```
routers
    pd.joe.aardvark
```

At this point, however, that identity hasn't made it onto the router yet. When you used `router-create` to name your new router, the server transmitted a wealth of information. To get that information to the router you need to know the host and port of the device. When running locally,

```
paradrop router-provision aardvark localhost 14231
```

To see the logs of your router while its running, try:

```
paradrop logs aardvark
```

But be warned! Currently they'll only respond if the router is awake.

CHAPTER 6

The Paradrop Instance System

This section focuses on the *Instance Tools*. This is the set of daemons and tools required to allow the Paradrop platform to function on virtual machines and real hardware. Use the information below to learn about Snappy Ubuntu and how we leverage it to create next generation smart routers.

Contents:

6.1 Build System

Paradrop includes a set of build tools to make development as easy as possible.

Currently this system takes the form of a bash script that automates installation and execution, but in time this may evolve into a published python package. This page outlines the steps required to manually build the components required to develop with paradrop.

Components in the build process:

- *Installing and running Ubuntu Snappy*
- *Building paradrop*
- *Installing paradrop*
- **'Creating chutes'_**

6.1.1 Installing and running Ubuntu Snappy

[Snappy](https://developer.ubuntu.com/en/snappy/) is an Ubuntu release focusing on low-overhead for a large set of platforms. These instructions are for getting a Snappy instance up and running using 'kvm'.

Download and unzip a snappy image:

```
wget http://releases.ubuntu.com/15.04/ubuntu-15.04-snappy-amd64-generic.img.xz
unxz ubuntu-15.04-snappy-amd64-generic.img.xz
```

Launch the snappy image using kvm:

```
kvm -m 512 -redir :8090::80 -redir :8022::22 ubuntu-15.04-snappy-amd64-generic.img
```

Connect to local instance using ssh:

```
ssh -p 8022 ubuntu@localhost
```

6.1.2 Building paradrop

Snappy is a closed system (by design!). Arbitrary program installation is not allowed, so to allow paradrop access to the wide world of `pypi` the build system relies on two tools.

- `virtualenv` is a tool that creates encapsulated environments in which python packages can be installed.
- `pex` can compress python packages into a zip file that can be executed by any python interpreter.

Dependancies for paradrop are packaged with the final snap as a pex file created by freezing a virtualenv. These are the steps needed to do this:

1. `venv.pex` is packaged with paradrop source code. This is a pex that contains only the virtualenv package. This file bootstraps virtualenv so it does not need to be installed on the local system.
2. A new virtual environment is created under `/buildenv/env` by calling `venv.pex ./buildenv/env`
3. The environment is activated with `source ./buildenv/env/bin/activate`. Any python package installations will now be placed here.
4. Paradrop is installed with `pip install -e .`. This installs paradrop in the virtual as well as all dependancies. Dependancies are listed in `src/setup.py`. You must add depedencies here in order to include new python packages with paradrop.
5. Dependancies are saved into `bin/pddepedencies.pex` with the command `pex -r docs/requirements.txt -o bin/pddepedencies.pex`. Note: requirements are written out to the file in step 4. This is done so that the `paradrop` dependancy is not included in the pex, since pex won't know how to look for it! The command used to do this is `pip freeze | grep -v 'pex' | grep -v 'paradrop' > docs/requirements.txt`.

At this point you can run paradrop by activating the virtualenv (step #3) and then simply calling `paradrop`. Note that the bundled dependancies pex does not affect locally running paradrop instances– its used in the next section.

6.1.3 Installing paradrop

All programs installed on snappy are called `snaps`. Snappy development tools are required to build snaps:

```
sudo add-apt-repository ppa:snappy-dev/tools
sudo apt-get update
sudo apt-get install snappy-tools bzr
```

To build a snap:

```
snappy build .
```

Push a snap to a running instance of snappy:

```
snappy-remote --url=ssh://localhost:8022 install SNAPNAME
```

6.2 Snappy Confinement

Snappy confines running applications in two ways: directory isolation and mandatory access control. Directory isolation means the application cannot leave its installed directory. MAC means the application cannot execute any system commands or access any files it does not have explicit, predetermined permissions to.

MAC is the more serious hurdle for paradrop development. Snaps declare permissions through an AppArmor profile.

6.2.1 Getting started with Profile Generation

Install tools and profiles:

```
sudo apt-get install apparmor-profiles apparmor-utils
```

List active profiles:

```
sudo apparmor_status
```

Profiles in complain mode log behavior, while those in enforce mode actively restrict it.sudo apt-get install apparmor-utils

The following steps assume paradrop is installed on the system and not on a virtualenv.

Create a new, blank profile:

```
cd /etc/apparmor.d/
sudo aa-autodep paradrop
```

Use aa-complain to put the profile in complain mode:

```
sudo aa-complain paradrop
```

Excercise the application! AppArmor will surreptitiously watch the program in the background and log all behavior. Once finished, use the following command to go through the resulting requests, approve or deny them, and autogenerate a profile:

```
sudo aa-logprof
```

6.3 Documentation and tests

Documentation is handled by sphinx and readthedocs.

Testing is a joint effort between nosetests, travis-ci, and coveralls.

6.3.1 Documentation

Information about docs creation, management, and display.

Sphinx reads files in reStructuredText and builds a set of HTML pages. Every time a new commit is pushed to github, readthedocs automatically updates documentation.

Additionally, sphinx knows all about python! The directives `automodule`, `autoclass`, `autofunction` and more instruct sphinx to inspect the code located in `src/` and build documentation from the docstrings within.

For example, the directive `.. automodule:: paradrop.backend` will build all the documentation for the given package. See google for more instructions.

All docstring documentation is rebuilt on every commit (unless there's a bug in the code.) Sphinx does not, however, know about structural changes in code! To alert sphinx of these changes, use the `autodoc` feature:

```
sphinx-apidoc -f -o docs/api paradrop/paradrop/
```

This scans packages in the `src/paradrop` directory and creates .rst files in `docs/api`. The root file `index.rst` links to `modules.rst`, connecting the newly generated api code with the main documentation.

To create the documentation locally, run:

```
cd docs
make html
python -m SimpleHTTPServer 9999
```

Open your web browser of choice and point it to http://localhost:9999/_build/html/index.html.

6.3.2 Testing

As mentioned above, all testing is automatically run by travis-ci, a continuous integration service.

To manually run tests, install nosetest:

```
pip install nose
```

Run all tests:

```
nosetests
```

Well thats easy. How does nose detect tests? All tests live in the `tests/` directory. Nose adheres to a simple principle: anything marked with `test` in its name is most likely a test. When writing tests, make sure all functions begin with `test`.

Coverage analysis detects how much of the code is used by a test suite. If the result of the coverage is less than 100%, someone slacked. Install coveralls:

```
pip install coveralls
```

Run tests with coverage analysis:

```
nosetests --with-coverage --cover-package=paradrop
```

CHAPTER 7

Architecture

This section details some of the non-obvious architectural features of paradrop.

This is a work in progress. Check back later!

CHAPTER 8

Known Issues

Please check here for issues during setup.

8.1 Issues using pdbuild.sh

These issues are related to the *Instance Tools* found on github.

8.1.1 Issue 1: `pdbuild.sh install` fails

Docker snap is missing inside of virtual router, run `pdbuild.sh install_deps`:

```
issues while running ssh command: Installing /tmp/paradrop_0.1.0_all.snap
2015/08/11 21:22:57 Signature check failed, but installing anyway as requested
/tmp/paradrop_0.1.0_all.snap failed to install: missing frameworks: docker
```

8.1.2 Issue 2: `pdbuild.sh install` fails

This is a known issue for the Paradrop team, if you get this please email us at developers@paradrop.io:

```
Installing paradrop_0.1.0_all.snap from local environment

issues while running ssh command: Installing /tmp/paradrop_0.1.0_all.snap
2015/08/11 21:29:48 Signature check failed, but installing anyway as requested
/tmp/paradrop_0.1.0_all.snap failed to install: [start paradrop_pdconfd_0.1.0.service]
failed with exit status 1: Job for paradrop_pdconfd_0.1.0.service failed.
See "systemctl status paradrop_pdconfd_0.1.0.service" and "journalctl -xe" for details.
```

8.1.3 Issue 3: `pdbuild.sh up` fails

This is very common and will happen if you delete your VM and setup a fresh one, the solution is simple and is stated in the error message:

```
Failed to setup keys: failed to setup keys: issues while running ssh command:
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
```

```
It is also possible that a host key has just been changed.
The fingerprint for the ECDSA key sent by the remote host is
e6:ec:b1:93:7d:91:84:50:19:36:14:8e:ce:ef:6a:0b.
Please contact your system administrator.
```

8.2 Issues using paradrop command (pdtools)

These issues are related to the *Build Tools* found on PyPI.

8.2.1 Issue 1: All `paradrop` commands fail

pdtools uses `enum34` rather than the `enum` package from PyPI, make sure you have the right one:

```
Traceback (most recent call last):
  File "/usr/local/bin/paradrop", line 7, in <module>
    from pdtools.main import main
  File "/usr/local/lib/python2.7/dist-packages/pdtools/__init__.py", line 1, in <module>
    from . import main
  File "/usr/local/lib/python2.7/dist-packages/pdtools/main.py", line 29, in <module>
    from pdtools.lib import output, riffle, names
  File "/usr/local/lib/python2.7/dist-packages/pdtools/lib/names.py", line 61, in <module>
    NameTypes.user: re.compile(r'^pd\.%s$' % n),
AttributeError: 'Enum' object has no attribute 'user'
```

CHAPTER 9

Frequently Asked Questions

Please check here for any FAQ's.

CHAPTER 10

paradrop

10.1 Subpackages

10.1.1 paradrop.backend package

Subpackages

paradrop.backend.exc package

Submodules

paradrop.backend.exc.executionplan module

abortPlans(*update*)

This function should be called if one of the Plan objects throws an Exception. It takes the PlanMap argument and calls the getNextAbort function just like executePlans does with todoPlans. This dynamically generates an abort plan list based on what plans were originally executed. Returns:

True in error : This is really bad False otherwise : we were able to restore system state back to before the executeplans function was called

aggregatePlans(*update*)

Takes the PlanMap provided which can be a combination of changes for multiple different chutes and it puts things into a sane order and removes duplicates where possible.

This keeps things like reloading networking from happening twice if 2 chutes make changes.

Returns: A new PlanMap that should be executed

executePlans(*update*)

Primary function that actually executes all the functions that were added to plans by all the exc modules. This function can heavily modify the OS/files/etc.. so the return value is very important. Returns:

True in error : abortPlans function should be called False otherwise : everything is OK

generatePlans(*update*)

For an update object provided this function references the updateModuleList which lets all exc modules determine if they need to add functions to change the state of the system when new chutes are added to the OS.

Returns: True in error, as in we should stop with this update plan

paradrop.backend.exc.files module

generateFilesPlan(*chuteStor*, *newChute*, *chutePlan*)

This function looks at a diff of the current Chute (in @chuteStor) and the @newChute, then adds Plan() calls to make the Chute match the @newChute.

Returns: None : means continue to pass this chute update to the rest of the chain. True : means stop updating, but its ok (no errors or anything) str : means stop updating, but some error occured (contained in the string)

generatePlans(*update*)

This function looks at a diff of the current Chute (in @chuteStor) and the @newChute, then adds Plan() calls to make the Chute match the @newChute.

Returns: True: abort the plan generation process

paradrop.backend.exc.name module

generatePlans(*update*)

This function looks at a diff of the current Chute (in @chuteStor) and the @newChute, then adds Plan() calls to make the Chute match the @newChute.

Returns: True: abort the plan generation process

paradrop.backend.exc.plangraph module

class Plan(*func*, **args*)

Helper class to hold onto the actual plan data associated with each plan

class PlanMap(*name*)

This class helps build a dependency graph required to determine what steps are required to update a Chute from a previous version of its configuration.

addMap(*other*)

Takes another PlanMap object and appends whatever the plans are into this plans object.

addPlans(*priority*, *todoPlan*, *abortPlan=[]*)

Adds new Plan objects into the list of plans for this PlanMap.

Arguments: @priority : The priority number (1 is done first, 99 done last - see PRIORITYFLAGS section at top of this file) @todoPlan : A tuple of (function, (args)), this is the function that completes the actual task requested

the args can either be a single variable, a tuple of variables, or None.

@abortPlan [A tuple of (function, (args)) or None. This is what should be called if a plan somewhere in the chain] fails and we need to undo the work we did here - this function is only called if a higher priority function fails (ie we were called, then something later on fails that would cause us to undo everything we did to setup/change the Chute).

getNextAbort()

Like an iterator function, it returns each element in the list of abort plans in order.

Returns: (function, args) : Each todo is returned just how the user first added it None : None is returned when there are no more todo's

getNextTodo()

Like an iterator function, it returns each element in the list of plans in order.

Returns: (function, args) : Each todo is returned just how the user first added it None : None is returned when there are no more todo's

registerSkip(*func*)
Register this function as one to skip execution on, if provided it shouldn't return the (func, args) tuple as a result from the getNextTodo function.

sort()
Sorts the plans based on priority.

paradrop.backend.exc.resource module

generatePlans(*update*)
This function looks at a diff of the current Chute (in @chuteStor) and the @newChute, then adds Plan() calls to make the Chute match the @newChute.

Returns: True: abort the plan generation process

generateResourcePlan(*chuteStor*, *newChute*, *chutePlan*)
This function looks at a diff of the current Chute (in @chuteStor) and the @newChute, then adds Plan() calls to make the Chute match the @newChute.

Returns: None : means continue to pass this chute update to the rest of the chain. True : means stop updating, but its ok (no errors or anything) str : means stop updating, but some error occured (contained in the string)

paradrop.backend.exc.runtime module

generatePlans(*update*)
This function looks at a diff of the current Chute (in @chuteStor) and the @newChute, then adds Plan() calls to make the Chute match the @newChute.

Returns: True: abort the plan generation process

paradrop.backend.exc.state module

generatePlans(*update*)
This function looks at a diff of the current Chute (in @chuteStor) and the @newChute, then adds Plan() calls to make the Chute match the @newChute.

Returns: True: abort the plan generation process

generateStatePlan(*chuteStor*, *newChute*, *chutePlan*)
This function looks at a diff of the current Chute (in @chuteStor) and the @newChute, then adds Plan() calls to make the Chute match the @newChute.

Returns: None : means continue to pass this chute update to the rest of the chain. True : means stop updating, but its ok (no errors or anything) str : means stop updating, but some error occured (contained in the string)

paradrop.backend.exc.struct module

generatePlans(*update*)
This function looks at a diff of the current Chute (in @chuteStor) and the @newChute, then adds Plan() calls to make the Chute match the @newChute.

Returns: True: abort the plan generation process

paradrop.backend.exc.traffic module

generatePlans(*update*)
This function looks at a diff of the current Chute (in @chuteStor) and the @newChute, then adds Plan() calls to make the Chute match the @newChute.

Returns: True: abort the plan generation process

generateTrafficPlan(*chuteStor*, *newChute*, *chutePlan*)
This function looks at a diff of the current Chute (in @chuteStor) and the @newChute, then adds Plan() calls to make the Chute match the @newChute.

Returns: None : means continue to pass this chute update to the rest of the chain. True : means stop updating, but its ok (no errors or anything) str : means stop updating, but some error occured (contained in the string)

Module contents

paradrop.backend.fc package

Submodules

paradrop.backend.fc.chutestorage module

class ChuteStorage(*filename=None*, *reactor=None*)
Bases: *paradrop.lib.utils.storage.PDStorage*

ChuteStorage class.

This class holds onto the list of Chutes on this AP.

It implements the PDStorage class which allows us to save the chuteList to disk transparently

attrSaveable()
Returns True if we should save the ChuteList, otherwise False.

chuteList = {}

clearChuteStorage()

deleteChute(*ch*)
Deletes a chute from the chute storage. Can be sent the chute object, or the chute name.

getAttr()
Get our attr (as class variable for all to see)

getChute(*name*)
Returns a reference to a chute we have in our cache, or None.

getChuteList()
Return a list of the GUIDs of the chutes we know of.

saveChute(*ch*)
Saves the chute provided in our internal chuteList. Also since we just received a new chute to hold onto we should save our ChuteList to disk.

setAttr(*attr*)
Save our attr however we want (as class variable for all to see)

paradrop.backend.fc.configurer module

class PDConfigurer(*storage*, *lclReactor*)
ParaDropConfigurer class. This class is in charge of making the configuration changes required on the chutes. It utilizes the ChuteStorage class to hold onto the chute data.

Use @updateChutes to make the configuration changes on the AP. This function is thread-safe, this class will only call one update set at a time. All others are held in a queue until the last update is complete.

clearUpdateList()
MUTEX: updateLock Clears all updates from list (new array).

getNextUpdate()
MUTEX: updateLock Returns the size of the local update queue.

performUpdates()
This is the main working function of the PDConfigurer class. It should be executed as a separate thread, it does the following:

checks for any updates to perform does them responds to the server removes the update checks for more updates

if more exist it calls itself again more quickly else it puts itself to sleep for a little while

updateList(***updateObj*)
MUTEX: updateLock Take the list of Chutes and push the list into a queue object, this object will then call the real update function in another thread so the function that called us is not blocked.

We take a callable responseFunction to call, when we are done with this update we should call it.

paradrop.backend.fc.updateObject module updateObject module.

This holds onto the UpdateObject class. It allows us to easily abstract away different update types and provide a uniform way to interpret the results through a set of basic actionable functions.

class UpdateChute(*obj*)
Bases: *paradrop.backend.fc.updateObject.UpdateObject*

Updates specifically tailored to chute actions like create, delete, etc...

saveState()
For chutes specifically we need to change the chuteStor object to reflect the new state of the system after a chute update. Perform that update here.

updateModuleList = [<module 'paradrop.backend.exc.name' from '/home/docs/checkouts/readthedocs.org/user_build

class UpdateObject(*obj*)
Bases: object

The base UpdateObject class, covers a few basic methods but otherwise all the intelligence exists in the inherited classes.

All update information passed by the API server is contained as variables of this class such as update.updateType, update.updateClass, etc...

By default, the following variables should be utilized: responses : an array of messages any module can choose to append warnings or errors to

failure [the module that chose to fail this update can set a string message to return] : to the user in the failure variable. It should be very clear as to why the : failure occurred, but if the user wants more information they may find it : in the responses variable which may contain debug information, etc...

complete(***kwargs*)
Signal to the API server that any action we need to perform is complete and the API server can finish its connection with the client that initiated the API request.

execute()
The function that actually walks through the main process required to create the chute. It follows the executeplan module through the paces of:

1.Generate the plans for each exc module

2.Prioritize the plans

3.Execute the plans

If at any point we fail then this function will directly take care of completing the update process with an error state and will close the API connection.

saveState()
Function should be overwritten for each UpdateObject subtype

updateModuleList = []

parse(*obj*)
Determines the update type and returns the proper class.

Module contents

paradrop.backend.pdconfd package

Subpackages

paradrop.backend.pdconfd.config package

Submodules

paradrop.backend.pdconfd.config.base module

class ConfigObject
Bases: object

addDependent(*dep*)

classmethod build(*manager*, *source*, *name*, *options*, *comment*)
Build a config object instance from the UCI section.

Arguments: source – file containing this configuration section name – name of the configuration section

If None, a unique name will be generated.

options – dictionary of options loaded from the section comment – comment string or None

commands(*allConfigs*)
Return a list of commands to execute.

Each one is a Command object.

getTypeAndName()
Return tuple (section type, section name).

lookup(*allConfigs*, *sectionType*, *sectionName*, *addDependent=True*)
Look up a section by type and name.

If addDependent is True (default), the current object will be added as a dependent of the found section.

Will raise an exception if the section is not found.

nextId = 0

options = []

optionsMatch(*other*)
Test equality of config sections by comparing option values.

typename = None

undoCommands (*allConfigs*)
Return a list of commands to execute.

Each one is a Command object.

paradrop.backend.pdconfd.config.command module

class Command (*priority*, *command*, *parent=None*)
Bases: object

PRIO_ADD_IPTABLES = 40

PRIO_CONFIG_IFACE = 20

PRIO_CREATE_IFACE = 10

PRIO_DELETE_IFACE = 50

PRIO_START_DAEMON = 30

execute()

success()
Returns True if the command was successfully executed.

paradrop.backend.pdconfd.config.dhcp module

class ConfigDhcp
Bases: *paradrop.backend.pdconfd.config.base.ConfigObject*

options = [{'default': None, 'required': True, 'type': <type 'str'>, 'name': 'interface'}, {'default': '12h', 'required': Tr

typename = 'dhcp'

class ConfigDnsmasq
Bases: *paradrop.backend.pdconfd.config.base.ConfigObject*

commands (*allConfigs*)

options = [{'default': None, 'required': False, 'type': <type 'list'>, 'name': 'interface'}, {'default': False, 'required': Fa

typename = 'dnsmasq'

undoCommands (*allConfigs*)

get_all_dhcp_interfaces (*allConfigs*)

paradrop.backend.pdconfd.config.firewall module

class ConfigRedirect
Bases: *paradrop.backend.pdconfd.config.base.ConfigObject*

commands (*allConfigs*)

options = [{'default': None, 'required': False, 'type': <type 'str'>, 'name': 'src'}, {'default': None, 'required': False, 't

typename = 'redirect'

undoCommands (*allConfigs*)

class ConfigZone
Bases: *paradrop.backend.pdconfd.config.base.ConfigObject*

commands (*allConfigs*)

interfaces (*allConfigs*)
List of interfaces in this zone (generator).

options = [{'default': None, 'required': True, 'type': <type 'str'>, 'name': 'name'}, {'default': None, 'required': False,

typename = 'zone'

undoCommands(*allConfigs*)

paradrop.backend.pdconfd.config.manager module

class **ConfigManager**(*writeDir*)

Bases: object

changingSet(*files*)

Return the sections from the current configuration that may have changed.

This checks which sections from the current configuration came from files in the given file list. These are sections that may be changed or removed when we reload the files.

execute(*commands*)

findMatchingConfig(*config*, *byName=False*)

Check the current config for an identical section.

Returns the matching object or None.

getPreviousCommands()

Get the most recent command list.

loadConfig(*search=None*, *execute=True*)

Load configuration files and apply changes to the system.

We process the configuration files in sections. Each section corresponds to an interface, firewall rule, DHCP server instance, etc. Each time we reload configuration files after the initial time, we check for changes against the current configuration. Here is the decision tree for handling differences in the newly loaded configuration vs. the existing configuration:

Section exists in current config (by type and name)?

- No -> Add section, apply changes, and stop.
- Yes -> Continue.

Section is identical to the one in the current config (by option values)?

- **No -> Revert current section, mark any affected dependents,** add new section, apply changes, and stop.
- Yes -> Continue.

Section has not changed but one of its dependencies has?

- No -> Stop.
- **Yes -> Revert current section, mark any affected dependents,** add new section, apply changes, and stop.

readConfig(*files*)

Load configuration files and return configuration objects.

This method only loads the configuration files without making any changes to the system and returns configuration objects as a generator.

statusString()

Return a JSON string representing status of the system.

The format will be a list of dictionaries. Each dictionary corresponds to a configuration block and contains at the following fields.

type: interface, wifi-device, etc. name: name of the section (may be autogenerated for some configs) comment: comment from the configuration file or None success: True if all setup commands succeeded

unload(*execute=True*)

waitSystemUp()
Wait for the first load to complete and return system status string.

findConfigFiles(*search=None*)
Look for and return a list of configuration files.

The behavior depends on whether the search argument is a file, a directory, or None.

If search is None, return a list of files in the system config directory. If search is a file name (not a path), look for it in the working directory first, and the system directory second. If search is a full path to a file, and it exists, then return that file. If search is a directory, return the files in that directory.

sortCommands(*commands*)
Return commands in order by priority.

The input should be a list of command objects. The output will be just the command part in order according to priority (ascending). For ties, the order from the original list is maintained.

paradrop.backend.pdconfd.config.network module

class ConfigInterface
Bases: *paradrop.backend.pdconfd.config.base.ConfigObject*

commands(*allConfigs*)

options = [{'default': None, 'required': True, 'type': <type 'str'>, 'name': 'proto'}, {'default': None, 'required': True,

typename = 'interface'

undoCommands(*allConfigs*)

paradrop.backend.pdconfd.config.wireless module

class ConfigWifiDevice
Bases: *paradrop.backend.pdconfd.config.base.ConfigObject*

commands(*allConfigs*)

options = [{'default': None, 'required': True, 'type': <type 'str'>, 'name': 'type'}, {'default': 1, 'required': True, 'type

typename = 'wifi-device'

class ConfigWifiIface
Bases: *paradrop.backend.pdconfd.config.base.ConfigObject*

commands(*allConfigs*)

options = [{'default': None, 'required': True, 'type': <type 'str'>, 'name': 'device'}, {'default': 'ap', 'required': True, '

typename = 'wifi-iface'

undoCommands(*allConfigs*)

isHexString(*data*)
Test if a string contains only hex digits.

Module contents

Submodules

paradrop.backend.pdconfd.client module

class Blocking(*deferred*)

Bases: object

Uses threading.Event to implement blocking on a twisted deferred object.

The wait method will wait for its completion and return its result.

Dear Lance. I hope you stub your toe.

unlock(*result*)

wait()

callDeferredMethod(**args*, ***kwargs*)

reload(*path*, *dbus=False*)

Reload file(s) specified by path.

This function blocks until the request completes. On completion it returns a status string, which is a JSON list of loaded configuration sections with a 'success' field. For critical errors such as failure to connect to the D-Bus service, it will return None.

reloadAll(*dbus=False*)

Reload all files from the system configuration directory.

This function blocks until the request completes. On completion it returns a status string, which is a JSON list of loaded configuration sections with a 'success' field. For critical errors such as failure to connect to the D-Bus service, it will return None.

waitSystemUp(*dbus=False*)

Wait for the configuration daemon to finish its first load.

This function blocks until the request completes. On completion it returns a status string, which is a JSON list of loaded configuration sections with a 'success' field. For critical errors such as failure to connect to the D-Bus service, it will return None.

paradrop.backend.pdconfd.main module This module listens for D-Bus messages and triggers reloading of configuration files. This module is the service side of the implementation. If you want to issue reload commands to the service, see the client.py file instead.

Operation:

- When triggered, read in UCI configuration files.
- Pass sections off to appropriate handlers (interface, etc.).
- Perform some validation (check for required options).
- Emit commands (start/stop daemon, ip, iw, etc.) into a queue.
- Issue commands, maybe rollback on failure.
- Update known state of the system.

Reference: http://excid3.com/blog/an-actually-decent-python-dbus-tutorial/

class ConfigService

Bases: txdbus.objects.DBusObject

configManager = None

dbusInterfaces = [<txdbus.interface.DBusInterface object at 0x7f8c2b9a4250>]

dbus_Reload(*name*)

dbus_ReloadAll()

dbus_Test()

dbus_UnloadAll()

dbus_WaitSystemUp()

listen(**args*, ***kwargs*)

run_pdconfd()
Start pdconfd daemon.

This enters the pdconfd main loop.

run_thread()
Start pdconfd service as a thread.

This function schedules pdconfd to run as a thread and returns immediately.

Module contents

paradrop.backend.pdfcd package

Submodules

paradrop.backend.pdfcd.apichute module

class ChuteAPI(*rest*)
The Chute API submodule. This class handles all API calls related to actionable items that directly effect chutes.

POST_createChute(*theSelf*, *request*, **args*, ***kwargs*)

POST_deleteChute(*theSelf*, *request*, **args*, ***kwargs*)

POST_startChute(*theSelf*, *request*, **args*, ***kwargs*)

POST_stopChute(*theSelf*, *request*, **args*, ***kwargs*)

paradrop.backend.pdfcd.apiinternal module

class Base(*module*, ***kwargs*)
Bases: `twisted.web.xmlrpc.XMLRPC`

lookupProcedure(*procedurePath*)

class ServerPerspective(*name*, *realm*)
Bases: `pdtools.lib.riffle.RifflePerspective`

destroy()

initialize()

perspective_subscribeLogs(**args*, ***kwargs*)
Fetch all logs since the target time. Stream all new logs to the server as they come in.

class ToolsPerspective(*name*, *realm*)
Bases: `pdtools.lib.riffle.RifflePerspective`

apiWrapper(*target*)
Add a final line of error and success callbacks before going onto the wire

api_provision(**args*, ***kwargs*)
Provision this router with an id and a set of keys.

This is a temporary call until the provisioning process is finalized.

castFailure(*failure*)
Converts an exception (or general failure) into an xmlrpc fault for transmission.

castSuccess(*res*)

checkStartRiffle()
Temporary function. Do not start serving or connecting over riffle until we have our keys (which occurs during currently optional provisioning)

pollServer(*host*)
Poll the server for a connection.

paradrop.backend.pdfcd.apiutils module backend.pdfcd.apiutils. Contains helper functions specific to the backend API code.

addressInNetwork(*ipaddr*, *netTuple*)
This function allows you to check if on IP belongs to a Network. Arguments:

unpacked IP address (use unpackIPAddr()) tuple of unpacked (addr, netmask) (use unpackIPAddrWithSlash())

Returns: True if in network False otherwise

calcDottedNetmask(*mask*)
Returns quad dot format of IP address.

getIP(*req*)
Returns the str IP addr from the request. NOTE: This is required because when we use nginx servers it is used as a proxy, so the REAL IP addr is stored in a HTTP header called 'X-Real-IP', so we need to check for this first, otherwise the request.getClientIP() is always going to return 'localhost' to us.

unpackIPAddr(*ip*)
Unpacks the 'IP' str. Returns a binary form of the ipaddr such that (ipaddr & netmask) will work.

unpackIPAddrWithSlash(*net*)
Unpacks the 'IP/bitmask' str. Returns a tuple of binary forms of both the ipaddr and netmask such that (ipaddr & netmask) will work.

paradrop.backend.pdfcd.server module pdfcd.server. Contains the classes required to establish a RESTful API server using Twisted.

class **ParadropAPIServer**(**args*, ***kwargs*)
Bases: *paradrop.lib.api.pdrest.APIResource*

The main API server module.

This sets up all of the submodules which should contain different types of RESTful API calls.

GET_test(*request*)
A Simple test method to ping if the API server is working properly.

complete(*update*)
Kicks off the properly threaded call to complete the API call that was passed to PDConfigurer. Since the PDConfigurer module has its own thread that runs outside of the main event loop, we have to call back into the event system properly in order to keep any issues of concurrency at bay.

default(*request*)
A dummy catch for all API calls that are not caught by any other module.

failprocess(*ip*, *request*, *logFailure*, *errorStmt*, *logUsage*, *errType*)
If logFailure is not None, Update the failureDict when the request does something wrong If logUsage is not None, log the usage track info.

Arguments:

ip : IP of client request : the request we received logFailure : If none, we do not log this failure to failure dict. Otherwise it is a tuple of

key : the key to use in the failureDict failureDict : the dict record the failure attempt history

errorStmt [A string to return to the user, looks like 'Malformed Body: %s'] so that we can add things like "Number of attempts remaining: 2" to the response

logUsage : A tuple of (tictoc and devid) used for usage tracker errorResponse: The error code to set response code

Returns: String to respond to the client

postprocess(*request*, *key*, *failureDict*, *logUsage*)
If the client is successful in their request, we should: * reset their failure attempts if failureDict is not none. * set success response code * If usage is not none, add usage track info of the api call

preprocess(*request*, *checkThresh*, *tictoc*)
Check if failure attempts for the user name has met the threshold. Arguments:

request : checkThresh : If None, no checking. Tuple of arguments for checking thresholds

ip: IP of client in string format token: sessionToken if found, or None username: username if signin, or None failureDict: the dict for failure history

ticktoc: If none, do not track the usage. The start time of the API call

Return: str: if threshold is met None: if ok

initializeSystem()
Perform some initialization steps such as writing important configuration.

setup(*args=None*)
This is the main setup function to establish the TCP listening logic for the API server. This code also takes into account development or unit test mode.

Module contents

Module contents

10.1.2 paradrop.lib package

Subpackages

paradrop.lib.api package

Submodules

paradrop.lib.api.pdapi module

exception PDAPIError(*etype*, *msg*)

Bases: exceptions.Exception

Exception class related to ParaDrop API calls.

getErrorToken()

Generates a random string which is used to match client issues with log output.

getResponse(*code*, **args*)

Designed to be called to provide the arguments for the Request.setResponseCode()

isPDError(*code*)

Checks all Paradrop API error codes, if the HTTP code is in our set it is assumed a PDAPI error.

paradrop.lib.api.pdrest module

ALL(*regex*)

APIDecorator(*admin=False*, *permission=None*, *requiredArgs=[]*, *optionalArgs=[]*)

The decorator for the API functions to make the API functions focus on their job. This decorator do the following things:

- Set HTTP header
- Get some common values like ip, tictoc
- Do the preprocess
- Extract arguments from HTTP body and put them into APIPackage
- Get devid if token is shown and put devid into APIPackage
- Check admin authorization if devid is shown and admin argument is set to be true
- Do the failprocess if fails. Do the postProcess if success

This decorator will pass an APIPackage to an API function and the API function is supposed to put return value into the API package Arguments:

- admin: if the function needs admin authority to call
- requiredArgs: the required arguments for this API, this wrapper will parse the required args from HTTP body and check if they exist in the body.
- optionalArgs: the optional arguments for this API, the args will be parsed from HTTP body
- permission: choose from None, "AP Owner", "Chute Owner"

TODO:

1.Permission

- multiple permission/multiple level of permission??
- More permissions: such as Vnet Owner, Group Owner

class APIPackage(*request*)

This is a class that wrap up the input and return value of API The input arguments will be in the inputArgs as a dict Result is True means the API return success Result is False means the API return failure Result is None means the API return NOT_YET_DONE

setFailure(*errType*, *errMsg=None*, *countFailure=True*)

setNotDoneYet()

setSuccess(*returnVal*)

class APIResource(**args*, ***kwargs*)
Bases: twisted.web.resource.Resource

getChild(*name*, *request*)

register(*method*, *regex*, *callback*)

unregister(*method=None*, *regex=None*, *callback=None*)

DELETE(*regex*)

GET(*regex*)

POST(*regex*)

PUT(*regex*)

maybeResource(*f*)

method_factory_factory(*method*)

Module contents

paradrop.lib.config package

Submodules

paradrop.lib.config.configservice module

configservice module: This module is responsible for "poking" the proper host OS services to change the host OS config. This would include things like changing the networking, DHCP server settings, wifi, etc..

reloadAll(*update*)

reloadQos(*update*)

paradrop.lib.config.devices module

Detect physical devices that can be used by chutes.

This module detects physical devices (for now just network interfaces) that can be used by chutes. This includes WAN interfaces for Internet connectivity and WiFi interfaces which can host APs.

It also makes sure certain entries exist in the system UCI files for these devices, for example "wifi-device" sections. These are shared between chutes, so they only need to be added when missing.

getSystemDevices(*update*)
Detect devices on the system.

Store device information in cache key "networkDevices".

isVirtual(*ifname*)
Test if an interface is a virtual one.

FIXME: This just tests for the presence of certain strings in the interface name, so it is not very robust.

isWAN(*ifname*)
Test if an interface is a WAN interface.

isWireless(*ifname*)
Test if an interface is a wireless device.

setConfig(*chuteName*, *sections*, *filepath*)

setSystemDevices(*update*)
Initialize system configuration files.

Creates basic sections that all chutes require such as the "wan" interface.

paradrop.lib.config.dhcp module

getVirtDHCPSettings(*update*)
Looks at the runtime rules the developer defined to see if they want a dhcp server. If so it generates the data and stores it into the chute cache key:virtDHCPSettings.

setVirtDHCPSettings(*update*)
Takes a list of tuples (config, opts) and saves it to the dhcp config file.

paradrop.lib.config.dockerconfig module

getVirtPreamble(*update*)

paradrop.lib.config.firewall module

findMatchingInterface(*iface_name*, *interfaces*)
Search an interface list for one matching a given name.

iface_name can contain shell-style wildcards (* and ?).

getDeveloperFirewallRules(*update*)
Generate other firewall rules requested by the developer such as redirects. The object returned is a list of tuples (config, options).

getOSFirewallRules(*update*)
There is a set of default things that must exist just for the chute to function at all, generate those here.

Stored in key: osFirewallRules

setOSFirewallRules(*update*)
Takes a list of tuples (config, opts) and saves it to the firewall config file.

paradrop.lib.config.network module

abortNetworkConfig(*update*)
Release resources claimed by chute network configuration.

getNetworkConfig(*update*)
For the Chute provided, return the dict object of a 100% filled out configuration set of network configuration. This would include determining what the IP addresses, interfaces names, etc...

getOSNetworkConfig(*update*)
Takes the network interface obj created by NetworkManager.getNetworkConfiguration and returns a properly formatted object to be passed to the OpenWrtConfig class. The object returned is a list of tuples (config, options).

getVirtNetworkConfig(*update*)

interfaceDefsEqual(*iface1*, *iface2*)

reclaimNetworkResources(*chute*)
Reclaim network resources for a previously running chute.

This function only applies to the special case in which pd starts up and loads a list of chutes that were running. This function marks their IP addresses and interface names as taken so that new chutes will not use the same values.

setOSNetworkConfig(*update*)
Takes a list of tuples (config, opts) and saves it to the network config file.

paradrop.lib.config.osconfig module

osconfig module: This module is in charge of changing configuration files for pdfcd on the host OS. This relates to things like network, dhcp, wifi, firewall changes. Pdfcd should be able to make simple abstracted calls into this module so that if we need to change what type of OS config we need to support only this module would change.

revertConfig(*update*, *theType*)
Basically the UCI system saves a backup of the original config file, if we need to revert changes at all, we can just tell our UCI module to revert back using that backup copy.

paradrop.lib.config.pool module

class NetworkPool(*network*, *subnetSize=24*)
Bases: *paradrop.lib.config.pool.ResourcePool*

class NumericPool(*digits=4*)
Bases: *paradrop.lib.config.pool.ResourcePool*

class ResourcePool(*values*, *numValues*)
Bases: object

next()

release(*item*)

reserve(*item*, *strict=True*)
Mark item as used.

If strict is True, raises an exception if the item is already used.

paradrop.lib.config.uciutils module

appendListItem(*options*, *name*, *value*)
Add a list item to UCI options.

The way we store lists for UCI options is rather bizarre, so this function takes care of that.

options: dictionary of options for a UCI section name: string name of the list option value: string value to append

removeConfigs(*chute*, *cacheKeys*, *filepath*)
used to modify config file of each various setting in /etc/config/

restoreConfigFile(*chute*, *configname*)
Restore a system config file from backup.

This can only be used during a chute update operation to revert changes that were made during that update operation.

configname: name of configuration file ("network", "wireless", etc.)

setConfig(*chute*, *old*, *cacheKeys*, *filepath*)
Helper function used to modify config file of each various setting in /etc/config/ Returns:

True: if it modified a file False: if it did NOT cause any modifications

setList(*options*, *name*, *values*)
Set a list item in UCI options.

The way we store lists for UCI options is rather bizarre, so this function takes care of that.

options: dictionary of options for a UCI section name: string name of the list option values: list of string values

paradrop.lib.config.wifi module
getOSWirelessConfig(*update*)
setOSWirelessConfig(*update*)

Module contents

paradrop.lib.utils package

Submodules

paradrop.lib.utils.addresses module
areWanPortsAvailable(*portRange*, *takenPorts*, *chuteStor*, *name*)
Make sure that if we are forwarding a wan port, we have not already forwarded it.
checkPhyExists(*radioid*)
Check if this chute exists at all, a directory /sys/class/ieee80211/phyX must exist.

getChuteIntf(*name*, *netIntfs*)
This function takes a network interface name, and parses through the netIntfs object provided looking for a match, it returns name if none is found.

Example: if 'lan' is the name, 'lan' will be returned. if 'wifilxc' is the name, 'c####wifilxc' will be returned.

This function also deals with the usage of macro expansions:

Example: If the developer defines a 'lan' interface for their chute, but they also have a rule that needs to point to the HOST lan interface, a conflict will occur.

To solve this, the developer would specify the HOST lan with '@net.host.lan' and the chute lan with 'lan'.

getGatewayIntf(*ch*)
Looks at the key:networkInterfaces for the chute and determines what the gateway should be including the IP address and the internal interface name.

Returns: A tuple (gatewayIP, gatewayInterface) None if networkInterfaces doesn't exist or there is an error

getInternalIntfList(*ch*)
Takes a chute object and uses the key:networkInterfaces to return a list of the internal network interfaces that will exist in the chute (e.g., eth0, eth1, ...)

Returns: A list of interface names None if networkInterfaces doesn't exist or there is an error

getSubnet(*ip1*, *sn1*)

getWANIntf(*ch*)
Looks at the key:networkInterfaces for the chute and finds the WAN interface.

Returns: The dict from networkInterfaces None

incIpaddr(*ipaddr*, *inc=1*)
Takes a quad dot format IP address string and adds the @inc value to it by converting it to a number.

Returns: Incremented quad dot IP string or None if error

isIpAvailable(*ipaddr*, *chuteStor*, *name*)
Make sure this IP address is available. Checks the IP addresses of all zones on all other chutes,

makes sure subnets are not the same.

isIpValid(*ipaddr*)
Return True if Valid, otherwise False.

isRadioPassedthrough(*radioid*, *chuteStor*, *name*)
Check if any chute has already passed through to a chute.

isSameSubnet(*ip1*, *ip2*)

isStaMeshAvailable(*chuteStor*, *name*)
Make sure this sta/mesh is available. Only one per device because we attach to wan directly.

isStaMeshOnRadio(*radioid*, *chuteStor*, *name*)
If we are a wifi chute, we cannot have an sta on this channel.

isStaticIpAvailable(*ipaddr*, *chuteStor*, *name*)
Make sure this static IP address is available. Checks the IP addresses of all zones on all other chutes,

makes sure not equal.

isWifiIntAvailable(*radioid*, *numWifi*, *chuteStor*, *name*)
Make sure that fewer than 7 wifi interfaces have been configured. Otherwise, the wireless card will fail.

isWifiOnRadio(*radioid*, *chuteStor*, *name*)
Make sure this sta is available. Only one per device because we attach to wan directly.

isWifiSSIDAvailable(*ssid*, *chuteStor*, *name*)
Make sure this SSID is available.

maxIpaddr(*ipaddr*)
Takes a quad dot format IP address string and makes it the largest valid value still in the same subnet.

Returns: Incremented quad dot IP string or None if error

paradrop.lib.utils.dockerapi module Functions associated with deploying and cleaning up docker containers.

build_host_config(*update*)
Build the host_config dict for a docker container based on the passed in update.

Parameters **update** (*obj*) – The update object containing information about the chute.

Returns (dict) The host_config dict which docker needs in order to create the container.

failAndCleanUpDocker(*validImages*, *validContainers*)
Clean up any intermediate containers that may have resulted from a failure and throw an Exception so that the abort process is called.

Parameters

- **validImages** (*list*) – A list of dicts containing the Id's of all the images that should exist on the system.
- **validContainers** (*list*) – A list of the Id's of all the containers that should exist on the system.

Returns None

removeChute(*update*)
Remove a docker container and the image it was built on based on the passed in update.

Parameters **update** (*obj*) – The update object containing information about the chute.

Returns None

restartChute(*update*)
Start a docker container based on the passed in update.

Parameters **update** (*obj*) – The update object containing information about the chute.

Returns None

setup_net_interfaces(*update*)
Link interfaces in the host to the internal interface in the docker container using pipework.

Parameters **update** (*obj*) – The update object containing information about the chute.

Returns None

startChute(*update*)
Build and deploy a docker container based on the passed in update.

Parameters **update** (*obj*) – The update object containing information about the chute.

Returns None

stopChute(*update*)
Stop a docker container based on the passed in update.

Parameters **update** (*obj*) – The update object containing information about the chute.

Returns None

writeDockerConfig()
Write options to Docker configuration.

Mainly, we want to tell Docker not to start containers automatically on system boot.

paradrop.lib.utils.pdos module

basename(*x*)
copy(*a*, *b*)

copytree(*a*, *b*)
shutil's copytree is dumb so use distutils.

doMount(*part*, *mnt*)
This function mounts @part to @mnt.

doUnmount(*mnt*)
This function unmounts @mnt.

exists(*p*)

fixpath(*p*)
This function is required because if we need to pass a path to something like tarfile, we cannot overwrite the function to fix the path, so we need to expose it somehow.

getFileType(*f*)

getMountCmd()

isMount(*mnt*)
This function checks if @mnt is actually mounted.

isdir(*a*)

isfile(*a*)

ismount(*p*)

makeExecutable(**args*)
The function that takes the list of files provided and sets the X bit on them.

mkdir(*p*)

move(*a*, *b*)

open(*p*, *mode*)

oscall(*cmd*, *get=False*)
This function performs a OS subprocess call. All output is thrown away unless an error has occured or if @get is True Arguments:

@cmd: the string command to run [get] : True means return (stdout, stderr)

Returns: None if not @get and no error (stdout, retcode, stderr) if @get or yes error

readFile(*filename*, *array=True*, *delimiter='\n'*)
Reads in a file, the contents is NOT expected to be binary. Arguments:

@filename: absolute path to file @array : optional: return as array if true, return as string if False @delimiter: optional: if returning as a string, this str specifies what to use to join the lines

Returns: A list of strings, separated by newlines None: if the file doesn't exist

remove(*a*)

symlink(*a*, *b*)

syncFS()

unlink(*p*)

write(*filename*, *data*, *mode='w'*)
Writes out a config file to the specified location.

writeFile(*filename*, *line*, *mode='a'*)
Adds the following cfg (either str or list(str)) to this Chute's current config file (just stored locally, not written to file.

paradrop.lib.utils.pdosq module Quiet pdos module. Implements utility OS operations without relying on the output module. Therefore, this module can be used by output without circular dependency.

makedirs(*p*)
Recursive directory creation (like mkdir -p). Returns True if the path is successfully created, False if it existed already, and raises an OSError on other error conditions.

paradrop.lib.utils.restart module lib.utils.restart Contains the functions required to restart chutes properly on power cycle of device. Checks with pdconfd to make sure it was able to properly bring up all interfaces before starting chutes.

reloadChutes()
This function is called to restart any chutes that were running prior to the system being restarted. It waits for pdconfd to come up and report whether or not it failed to bring up any of the interfaces that existed before the power cycle. If pdconfd indicates something failed we then force a stop update in order to bring down all

interfaces associated with that chute and mark it with a warning. If the stop fails we mark the chute with a warning manually and change its state to stopped and save to storage this isn't great as it could mean our system still has interfaces up associated with that chute. If pdconfd doesn't report failure we attempt to start the chute and if this fails we trust the abort process to restore the system to a consistent state and we manually mark the chute as stopped and add a warning to it. :param None :returns: (list) A list of update dicts to be used to create updateObjects that should be run before accepting new updates.

updateStatus(*update*)
This function is a callback for the updates we do upon restarting the system. It checks whether or not the update completed successfully and if not it changes the state of the chute to stopped and adds a warning. :param update: The update object containing information about the chute that was created and whether it was successful or not. :type update: obj :returns: None

paradrop.lib.utils.storage module
class PDStorage(*filename*, *reactor*, *saveTimer*)
ParaDropStorage class.

This class is designed to be implemented by other classes. Its purpose is to make whatever data is considered important persistant to disk.

This is done by providing a reactor so a "LoopingCall" can be utilized to save to disk.

The implementer can override functions in order to implement this class: getAttr() : Get the attr we need to save to disk setAttr() : Set the attr we got from disk importAttr(): Takes a payload and returns the properly formatted data exportAttr(): Takes the data and returns a payload attrSaveable(): Returns True if we should save this attr

attrSaveable()
THIS SHOULD BE OVERRIDEN BY THE IMPLEMENTER.

exportAttr(*data*)
By default do nothing, but expect that this function could be overwritten

importAttr(*pyld*)
By default do nothing, but expect that this function could be overwritten

loadFromDisk()
Attempts to load the data from disk. Returns True if success, False otherwise.

saveToDisk()
Saves the data to disk.

paradrop.lib.utils.uci module
class UCIConfig(*filepath*)

Wrapper around the UCI configuration files. These files are found under /etc/config/*, and are used by OpenWrt to keep track of configuration for modules typically found in /etc/init.d/*

The modules of interest and with current support are:

- firewall
- network
- wireless
- qos

- This class should work with any UCI module but ALL others are UNTESTED!

New configuration settings can be added to the UCI file via addConfig().

Each UCI config file is expected to contain the following syntax:

config keyA [valueA] option key1 value1 ... list key2 value1 list key2 value2 ... list key3 value1 list key3 value2

Based on the UCI file above, the config syntax would look like the following: config is a list of tuples, containing 2 dict objects in each tuple:

- **tuple[0] describes the first line (config keyA [valueA])** {'type': keyA, 'name': valueA} The value parameter is optional and if missing, then the 'name' key is also missing (rather than set to None).
- **tuple[1] describes the options associated with the settings (both 'option' and 'list' lines)** {'key1': 'value1', ...}

  If a list is present, it looks like the following:

  { ..., 'list': {

  'key2': [value1, value2, ...], 'key3': [value1, value2, ...] }

  }

So for the example above, the full config definition would look like: C = {'type': 'keyA', 'name': 'valueA'} O = {'key1': 'value1', 'list': {'key2': ['value1', 'value2'], 'key3': ['value1', 'value2']}} config = [(C, O)]

addConfig(*config*, *options*)
Adds the tuple to our config.

addConfigs(*configs*)
Adds a list of tuples to our config

backup(*backupToken*)
Puts a backup of this config to the location specified in @backupPath.

checkWanConfig(*internalid*)

delConfig(*config*, *options*)
Finds a match to the config input and removes it from the internal config data structure.

delConfigs(*configs*)
Adds a list of tuples to our config

existsConfig(*config*, *options*)
Tests if the (config, options) is in the current config file.

getChuteConfigs(*internalid*)

getConfig(*config*)
Returns a list of call configs with the given title

getConfigIgnoreComments(*config*)
Returns a list of call configs with the given title. Comments are ignored.

readConfig()
Reads in the config file.

restore(*backupToken*, *saveBackup=True*)
Replaces real file (at /etc/config/*) with backup copy from /tmp/*-@backupToken location.

Arguments: backupToken: The backup token appended at the end of the backup path saveBackup : A flag to keep a backup copy or delete it (default is keep backup)

save(*backupToken=None*, *internalid=None*)
Saves out the file in the proper format.

Arguments:

[backupPath] [Save a backup copy of the UCI file to the path provided.] Should be a token name like 'backup', it gets appended with a hyphen.

chuteConfigsMatch(*chutePre*, *chutePost*)
Takes two lists of objects, and returns whether or not they are identical.

configsMatch(*a*, *b*)
Takes 2 config objects, returns True if they match, False otherwise.

getSystemConfigDir()

getSystemPath(*filename*)
Get the path to the system configuration file.

This function also attempts to create the configuration directory if it does not exist.

Typical filenames: network, wireless, qos, firewall, dhcp, etc.

isMatch(*a*, *b*)

isMatchIgnoreComments(*a*, *b*)

setupArgParse()

singleConfigMatches(*a*, *b*)

stringify(*a*)

Module contents

Submodules

paradrop.lib.chute module

class Chute(*descriptor*, *strip=None*)
Bases: object

Wrapper class for Chute objects.

appendCache(*key*, *val*)
Finds the key they requested and appends the val into it, this function assumes the cache object is of list type, if the key hasn't been defined yet then it will set it to an empty list.

delCache(*key*)
Delete the key:val from the _cache dict object.

dumpCache()
Return a string of the contents of this chute's cache. In case of catastrophic failure dump all cache content so we can debug.

fullDump()
Return a dump of EVERYTHING in this chute including all API data and all internal cache data.

getCache(*key*)
Get the val out of the _cache dict object, or None if it doesn't exist.

isValid()
Return True only if the Chute object we have has all the proper things defined to be in a valid state.

setCache(*key*, *val*)
Set the key:val into the _cache dict object to carry around.

paradrop.lib.settings module

This file contains any settings required by ANY and ALL modules of the paradrop system. They are defaulted to some particular value and can be called by any module in the paradrop system with the following code:

from paradrop import settings print(settings.STUFF)

These settings can be overriden by a file defined which contains the following syntax:

This changes a string default setting EXACT_SETTING_NAME0 "new string setting"

This changes a int default setting EXACT_SETTING_NAME1 int0

If settings need to be changed, they should be done so by the initialization code (such as pdfcd, pdapi_server, pdfc_config, etc...)

This is done by calling the following function: settings.updateSettings(filepath)

addSetting(*key*, *value*)
Adds a new setting to this module so other modules can see it.
Parameters
- **key** (*string.*) – the setting name.
- **value** (*variable.*) – the value of the setting.

Returns None

parseValue(*key*)
Attempts to parse the key value, so if the string is 'False' it will parse a boolean false.
Parameters **key** (*string*) – the key to parse

Returns the parsed key.

updateSettings(*slist=[]*)
Take a list of key:value pairs, and replace any setting defined. Also search through the settings module and see if any matching environment variables exist to replace as well.
Parameters **slist** (*array.*) – the list of key:val settings

Returns None

Module contents

10.2 Submodules

10.3 paradrop.main module

Core module. Contains the entry point into Paradrop and establishes all other modules. Does not implement any behavior itself.

class Nexus(*mode*)
Bases: pdtools.lib.nexus.NexusBase

connect(**args*, ***kwargs*)
Continuously tries to connect to server

onStart()

onStop()

serverConnected(*avatar*, *realm*)

serverDisconnected(*avatar*, *realm*)

main()

10.4 Module contents

CHAPTER 11

Paradrop

Paradrop is a software platform that enables apps to run on Wi-Fi routers. We call these apps “chutes” like a parachute. The name Paradrop comes from the fact that we are enabling the ability to “drop” supplies and resources (“apps”) into a difficult and not well-travelled environment - the home.

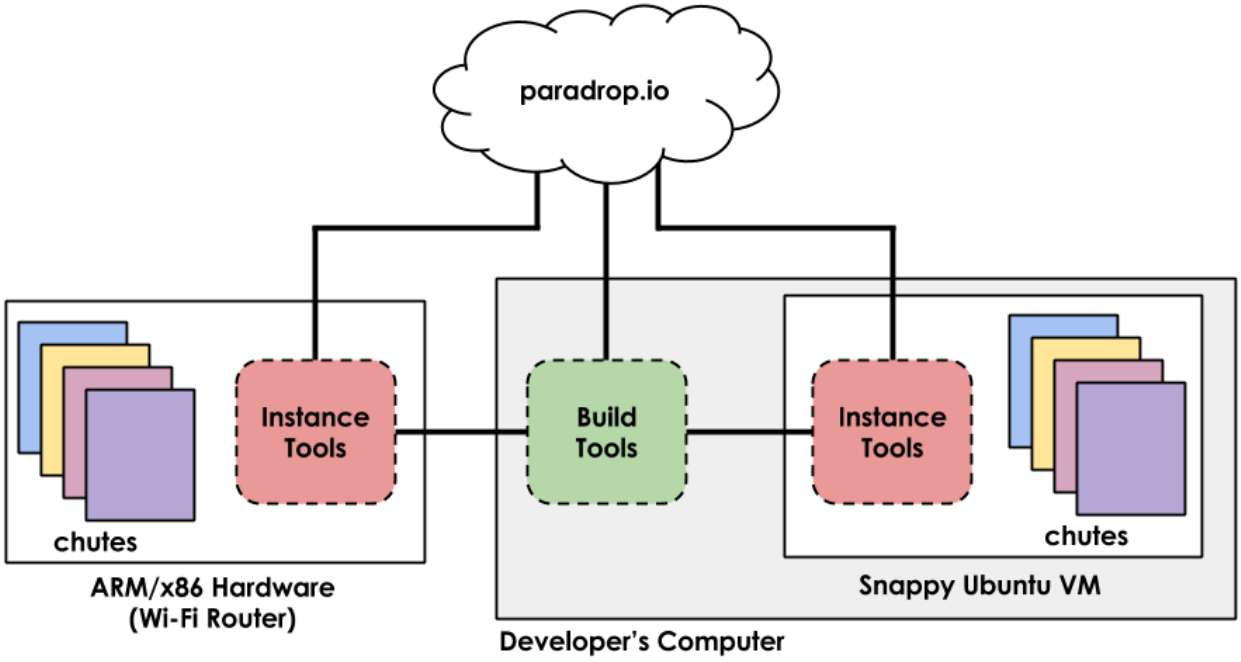
Our early versions of Paradrop relied on OpenWrt, however we are revamping the platform and tailoring it towards a broader developer community. Paradrop now runs on top of Snappy Ubuntu, a trimmed-down and secured operating system that can run on ARM and x86. We also enable our apps through containerization by leveraging Docker.

CHAPTER 12

The Paradrop workflow

There are two components to the Paradrop platform:

- The build tools - our CLI that enables registration, login, and control.
- The instance tools - our configuration daemons and tools to launch apps on hardware.



As you can see from the image above, we will refer to *Build Tools* when we talk about the CLI program running on your development computer that controls and communicates with the rest of the Paradrop platform. Treat this tool as your window into the rest of the Paradrop world. Our *Instance Tools* leverage programs like Docker to allow Paradrop apps to run on router hardware. This "hardware" could be a Raspberry Pi, or even a virtual machine on your computer that acts as a router (which is why we call it an Instance sometimes). Using Paradrop, you can actually plug in a USB Wi-Fi adapter and turn a virtual machine on your computer into a real router with our platform!

CHAPTER 13

Getting Started

Please visit the Getting Started page for a quick introduction to Paradrop.

CHAPTER 14

Where to go from here?

We have advanced app examples found under Apps on Paradrop. If you are interested in working on the instance side of paradrop (our github code) than check out: The Paradrop Instance System.

CHAPTER 15

What if I don't have Ubuntu?

We will soon switch our development system to Vagrant, which will allow support across all major operating systems. We will also update the docs with notes on how to download precompiled versions of our Paradrop instance tools once they have been fully tested.

p

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

U

W