# paradrop Documentation

***Release 0.10.2***

**Paradrop Labs**

January 30, 2018

Contents

ParaDrop is a platform for edge computing. This is best understood by comparison with the popular paradigm of cloud computing.

CHAPTER 1

# Cloud computing vs. edge computing

Cloud computing platforms such as Amazon EC2, Microsoft Azure, and Google Cloud Platform have grown in popularity as solutions for providing ubiquitous access to services across different user devices. Cloud computing has benefits for infrastructure providers, service providers, and end users. Infrastructure providers, i.e., cloud platform providers, take advantage of the economies of scale by managing and operating resources in a centralized manner. Cloud computing also provides reliable, scalable, and elastic resources to service providers. In addition, end users can access high-performance computing and large storage resources anywhere with Internet access at any time thanks to the cloud computing.

Despite all of the benefits of cloud computing, there are some inherent trade-offs in the approach. Cloud computing requires developers to host services and data on off-site data centers. That means that the computing and storage resources are spatially distant from end-users and out of their control, which raises issues related to network latency, security, and privacy. A growing number of high-quality services can benefit from computational tasks running closer to end-users, especially within their own home or office. By moving the computation closer to the users, at the edge of the network, services can take advantage of the lower latency to provide better responsiveness and user experience as well as conserve network bandwidth.

CHAPTER 2

# Where is the vantage point for edge computing?

There are various options for placing edge computing nodes within the network. ParaDrop takes the approach of placing the edge computing substrate within the WiFi access points (APs). The WiFi AP is uniquely suitable for edge computing for multiple reasons:

- WiFi APs are ubiquitous in homes and businesses and inexpensive to replace.
- WiFi APs are always on and available.
- WiFi APs reside directly on the data path between Internet services and end users.

CHAPTER 3

# How does it work?

ParaDrop is a research effort to build a highly programmable edge computing platform. The name for the project draws inspiration from the military use case of airdropping resources into the battlefield wherever they are needed most. Similarly, ParaDrop enables users and developers to *paradrop* edge services into the edge of the network as needed. Based on previous research work exploring the advantages of edge computing, we focus on building a platform that is friendly to both users and developers alike.

ParaDrop provides a similar runtime environment as the cloud computing platform to developers so that developers can easily port their services from the cloud to ParaDrop in part or in whole. It does this through lightweight containerization powered by Docker, which is already immensely popular in the cloud computing space. Containers allow developers the flexibility to build services with the programming languages, libraries, and frameworks they prefer, while being less resource-intensive than virtual machines. On top of that, ParaDrop offers a well-defined API that developers can leverage to implement and deploy interesting capabilities that are only available at the edge.

CHAPTER 4

# System Architecture

This section describes some of the important architectural features of ParaDrop. Our discussion will cover three major aspects of the ParaDrop design.

- The ParaDrop router
- The ParaDrop cloud controller
- The ParaDrop API

## ParaDrop router

The ParaDrop router is the key part of the ParaDrop platform. In addition to being a WiFi access point, it provides the substrate on which edge computing services can be deployed. We have built the reference ParaDrop routers based on the PC Engines APU and APU2 single board computer. The image below shows a router built with a PC Engines APU board.

In addition to the PC Engines APU2, the ParaDrop software implementation can be run on various other hardware platforms as well as virtual machines. Please refer to Hardware Support for more information about hardware setup for ParaDrop routers.

## ParaDrop cloud controller

The ParaDrop cloud controller is hosted at paradrop.org and provides a central location for tracking and managing ParaDrop routers as well as a Chute Store for software distribution. Users can sign up for a free account. For end users and administrators, it provides a dashboard to configure and monitor the ParaDrop routers under their control. The dashboard enables users to manage the services (called *chutes*) running on their routers. For developers, it provides the interface through which applications can be registered as ParaDrop chutes available for installation on routers.

## ParaDrop API

ParaDrop exports the platform's full capability through an API. Based on the functionality and location, the API can be divided into two parts: the cloud API and the edge API. The cloud API provides the management interfaces for applications to orchestrate the chutes from the cloud. Examples include resource permission management, chute deployment and management, and router configuration management. The edge API exports the local context information of the routers to the chutes to do some useful things locally. Examples include local wireless channel information and local wireless peripheral device access.

CHAPTER 5

# Quick Start

This section goes through the steps to create a ParaDrop account, activate a ParaDrop router, and install a hello-world chute on the router.

If you have received a device with ParaDrop already installed, you can start here. If you do not have a ParaDrop-enabled device, please visit the Hardware Support section.

## Create a ParaDrop account

With a ParaDrop account, users can manage the resources of ParaDrop through a web frontend.

1. Signup at https://paradrop.org/signup. You will receive a confirmation email from paradrop.org after you finish the signup.
2. Confirm your registration in the email.

## Boot the router

1. Using an Ethernet cable, connect the WAN port of the ParaDrop router to a modem, switch, or other device with access to the Internet.
2. Connect the power supply. To avoid malfunctioning due to arcing, it is recommended to connect the barrel connector to DC jack on the back of the router first and connect the adapter to a power outlet second.
3. Allow the router 1-2 minutes to start up, especially on the first boot.
4. Connect a device (laptop, phone, etc.) either to one of the LAN ports on the back of the router or to its WiFi network. Typically, the router will be preconfigured with an open ESSID called "ParaDrop". If the WiFi network has a password, that information will be provided separately.

## Activate a ParaDrop router

Activation associates the router with your account on paradrop.org so that you can manage the router and chutes through the cloud controller.

1. Login to paradrop.org.

2. Navigate to the Routers List page. If your router came with a Claim Token, enter that here and skip steps 3-5. Otherwise if you do not have a Claim Token, click Create Router. Give your router a unique name and an optional description to help you remember it and click Submit.
3. On the router page, find the "Router ID" and "Password" fields. You will need to copy this information to the router so that it can connect to the controller.
4. Open the router portal at http://<router_ip_address> (or http://paradrop.io if you are connected to the LAN port of the router or its WiFi network). You may be prompted for a username and password. The default login is "paradrop" with an empty password.
5. Click the "Activate the router with a ParaDrop Router ID" button and enter the information from the paradrop.org router page. If the activation was successful, you should see checkmarks appear on the "WAMP Router" and "ParaDrop Server" lines. You may need to refresh the page to see the update.
6. After you activate your router, you will see the router status is online at https://paradrop.org/routers.

## Install a hello-world chute

1. Make sure you have an activated, online router.
2. Go to the Chute Store tab on paradrop.org. There you will find some public chutes such as the hello-world chute. You can also create your own chutes here.
3. Click on the hello-world chute, click Install, click your router's name to select it, and finally, click Install.
4. That will take you to the router page again where you can click the update item to monitor its progress. When the installation is complete, an entry will appear under the Chutes list.
5. The hello-world chute starts a webserver, which is accessible at http://<router-ip-address>/chutes/hello-world. Once the installation is complete, test it in a web browser.

CHAPTER 6

# Hardware Support

This section describes various hardware platforms that we support for running ParaDrop.

If this is your first time working with ParaDrop and you do not have access to any of the supported hardware platforms, we recommend starting with our pre-built virtual machine image.

With the hardware platform up and running, you are ready to activate the router in ParaDrop. The page Quick Start gives detailed information about that.

## Virtual Machine

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

*NOTE*: These instructions assume you are running Ubuntu. The steps to launch a virtual machine may be different for other environments.

### Environment setup

These steps wil download our router image and launch it a virtual machine.

1. Install required packages:

```
sudo apt-get install qemu-kvm
```

2. Download the latest build of the Paradrop disk image. https://paradrop.org/release/latest/paradrop-amd64.img.xz
3. Extract the image:

```
xz -d paradrop-amd64.img.xz
```

4. Launch the VM:

```
sudo kvm -m 1024 \
-netdev user,id=net0,hostfwd=tcp::8000-:8000,hostfwd=tcp::8022-:22,hostfwd=tcp::8080-:80 \
-device virtio-net-pci,netdev=net0 -drive file=paradrop-amd64.img,format=raw
```

## First Boot Setup

After starting the virtual machine for the first time, follow the instructions on the screen. When it prompts for an email address, enter *info@paradrop.io*. This sets up a user account on the router called *paradrop* and prepares the router to receive software upgrades. Allow the router 1-2 minutes to complete its setup before proceeding.

Please note: there is no username/password to log into the system console. Please follow the steps in the next sections to access your router through paradrop.org or through SSH.

## Connecting to your Router with SSH

The router is running an SSH server, which is forwarded from localhost port 8022 with the kvm command above. The router does not accept password login by default, so you will need to have an RSA key pair available, and you can use the router configuration page to upload the public key and authorize it.

1. Open tools page on the router (http://localhost:8080/#!/tools).
2. Find the SSH Keys section and use the text area to submit your public key. Typically, your public key file will be found at *~/.ssh/id_rsa.pub*. You can use *ssh-keygen* to generate one if you do not already have one. Copy the text from the file, and make sure the format resembles the example before submitting.
3. After the key has been accepted by the router, you can login with the command *ssh -p 8022 paradrop@localhost*.

## Managing Virtual Machines Using virt-manager

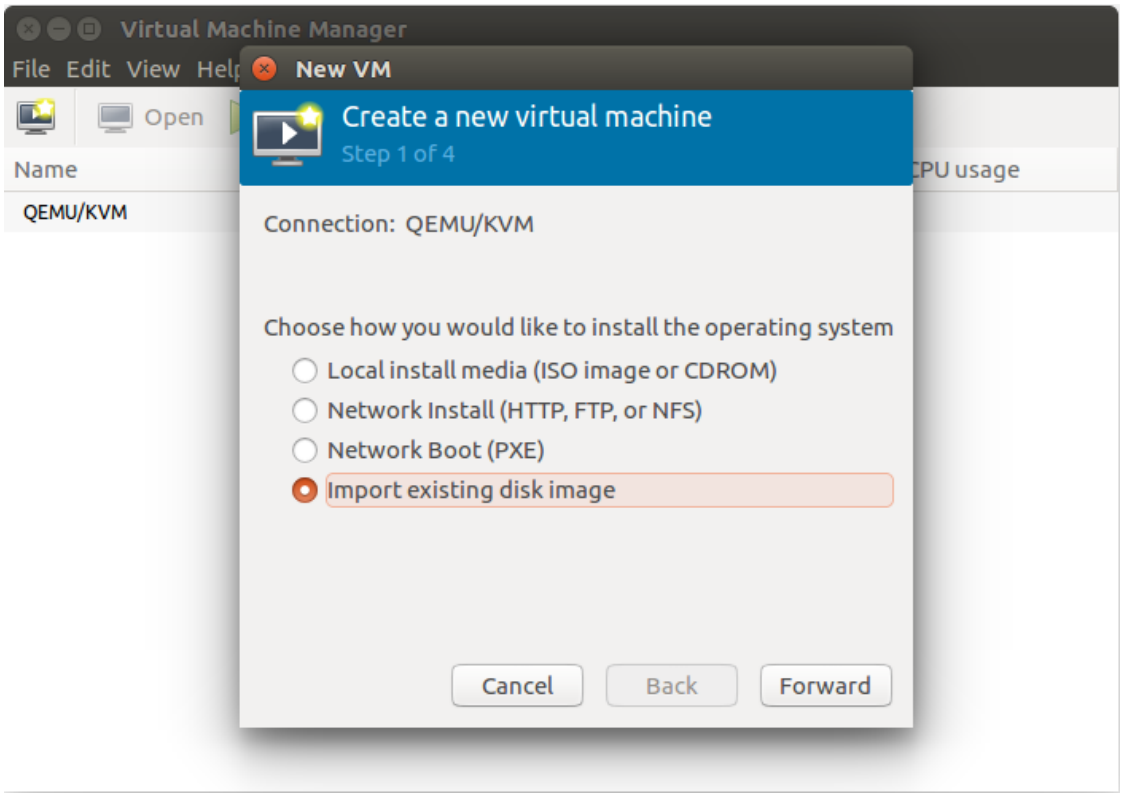
Even though many developers prefer command line tools to manage virtual machines, some developers likes to use GUI tools. In addition, GUI tools are convenient to support some advanced features, e.g., assigning some peripheral devices (USB WiFi dongle) from host to virtual machines. We recommend using "virt-manager" to run ParaDrop virtual machines. If you have not installed it on Ubuntu, you can use below command to install it:

```
sudo apt-get install virt-manager
```

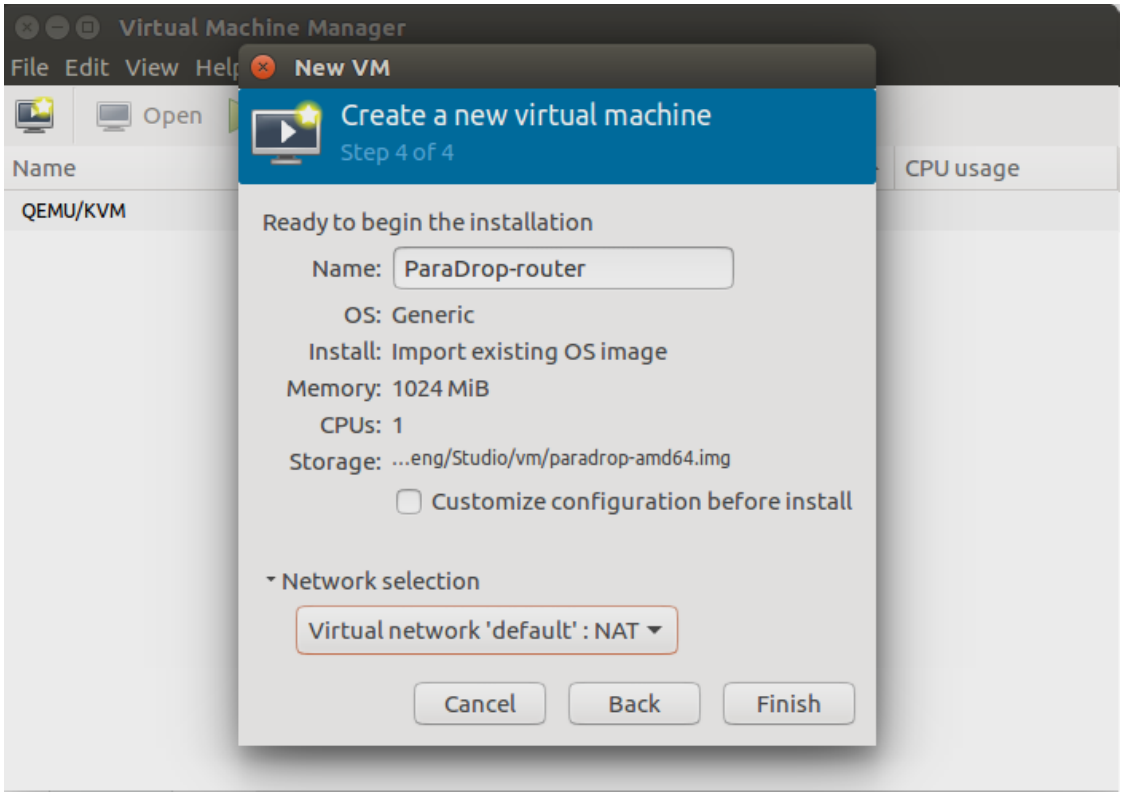
Then we can start virt-manager with below command:

```
sudo virt-manager
```

We can create a VM with the ParaDrop disk image.




Below is the configuration of the VM.




After that, we can boot the VM and configure the first boot as we do when run the VM with command line tools. However, the VM will have an IP address 192.168.122.x, so we can access http://<IP address of the VM> to access

the portal to upload ssh keys, and then login to it directly with the IP address.

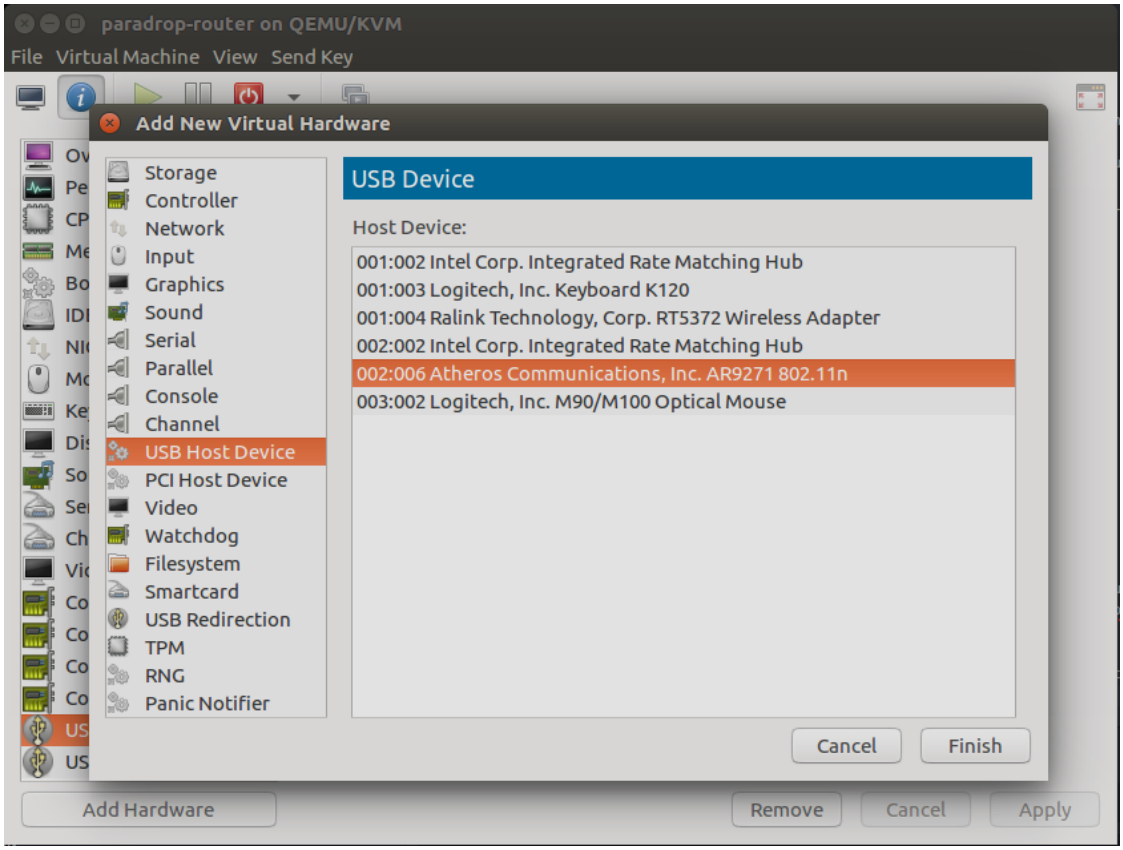
We can assign the USB WiFi dongle from the Host to the ParaDrop VM so that the ParaDrop running on the VM can support features related to WiFi. Before we do that, we need to disable the WiFi device for Host. We can do that with "rflist" command. Run below command to get the number of the WiFi device:

```
rflist list
```

Suppose the number of the WiFi device we want to assign to the ParaDrop VM is 2, then run below command to disable it for host OS:

```
rflist block 2
```

Then we can add the USB WiFi dongle to the VM.




We can run below command in ParaDrop VM to verify that the WiFi device has been detected:

```
iw dev
```

Sometimes, we have to repeat above steps to make sure the WiFi device can be used by the ParaDrop VM.

# PC Engines APU2

## Hardware requirements

- 1x system board (apu2c4)
- 1x case (case1d2u)

- 1-2x miniPCIe Wi-Fi modules (wle200nx for 802.11n or wle600vx for 802.11ac)
- 2-4x pigtails (pigsma)
- 2-4x antennas (antsmadb)
- 1x power supply (ac12vus2)
- 1x storage module (sd4b) or alternative (see below)

### Storage Module

The APU can boot from an SD card or an m-SATA SSD. These instructions are written assuming you will use an SD card because they are easier to flash from another machine. However, we do frequently build Paradrop routers with SSDs to take advantage of the higher storage capacity and read/write speeds. The 4GB pSLC module listed above is known to be very reliable, but you may also prefer a larger SD card.

### Preparing the SD card

1. Download the latest build of the Paradrop disk image. https://paradrop.org/release/0.10/paradrop-amd64.img.xz
2. Insert the SD card into the machine you used to download the image and find the device node for the card. This is often "/dev/sdb", but please make sure, as the next command will overwrite the contents of whatever device you pass.
3. Copy the Paradrop image to the SD card.:

```
xzcat paradrop-amd64.img.xz | sudo dd of=<DEVICE> bs=32M status=progress; sync
```

4. Remove the SD card and proceed to assemble the router.

### First Boot

If you know the IP address of the router, e.g. because you have access to the DHCP server upstream from the router, then you can skip this step and proceed with activating the router as described in the section **:doc:'../manual/index'_**.

The first time you boot the Paradrop router, you can optionally connect a serial cable to complete the Ubuntu Core setup process. The default configuration is 9600 8N1. After the router boots, press Enter when prompted and follow the instructions on the console to configure Ubuntu Core. If you have an Ubuntu One account, you can enter the email address here. For consistency with the rest of the instructions, we recommend using the address *info@paradrop.io*. You will be able to manage your router and install chutes through paradrop.org either way.

Take note of the IP address displayed in the console. You will need this address for the next step, activating the router. For example, the message below indicates that the router has IP address 10.42.0.162.

```
Congratulations! This device is now registered to info@paradrop.io.

The next step is to log into the device via ssh:

ssh paradrop@10.42.0.162
```

## Intel NUC

These instructions will help you install the ParaDrop daemon on the Intel NUC platform. At the end of this process, you will have a system ready for installing chutes.

We have specifically tested this process on the Skull Canyon (NUC6i7KYK) platform, which we recommend for high performance edge-computing needs.

## Hardware and software requirements

- **Intel NUC Skull Canyon NUC6i7KYK**
  - The Intel NUC devices generally do not come with memory or storage pre-installed.
  - Memory: we recommend at least one 8 GB DDR4 SODIMM.
  - Storage: we have generally found one 16 GB SD card to be sufficient for our storage needs, but we recommend using one MX300 M.2 SSD card for the higher read and write speeds.
  - We recommend updating the BIOS on the NUC. Follow the instructions on the Intel support site.
- 2 USB 2.0 or 3.0 flash drives (each 4 GB minimum)
- A monitor with an HDMI interface
- A network connection with Internet access
- An Ubuntu Desktop 16.04.1 LTS image.
- A ParaDrop disk image.

## Preparing for installation

1. Download the Ubuntu Desktop image and prepare a bootable USB flash drive.
2. Download the ParaDrop disk image and copy the file to the second flash drive.

## Boot from the Live USB flash drive

1. Insert the Live USB Ubuntu Desktop flash drive in the NUC.
2. Start the NUC and push F10 to enter the boot menu.
3. Select the USB flash drive as a boot option.
4. Select "Try Ubuntu without installing".

## Flash ParaDrop

1. Once the system is ready, insert the second USB flash drive which contains the ParaDrop disk image.
2. Open a terminal and run the following command, where <disk label> is the name of the second USB flash drive. We recommend that you double-check that /dev/sda is the desired destination **before running dd**.

```
zcat /media/ubuntu/<disk label>/paradrop_router.img.gz | sudo dd of=/dev/sda bs=32M status=progr
```

3. Reboot the system and remove all USB flash drives when prompted to do so.

## First boot

1. At the Grub menu, press 'e' to edit the boot options.
2. Find the line that begins with "linux" and append the option "nomodeset". It should look like "linux (loop)/kernel.img $cmdline nomodeset". Adding this option will temporarily fix a graphics issue that is known to occur with the Intel NUC.
3. Press F10 to continue booting.
4. Press Enter when prompted, and follow the instructions on the screen to configure Ubuntu Core. If you have an Ubuntu One account. By connecting your Ubuntu One account, you will be able to login via SSH with the key(s) attached to your account. Otherwise, if you do not have an Ubuntu One account or do not wish to use it, you may enter "info@paradrop.io" as your email address. You will still be able to manage your router and install chutes through paradrop.org either way.
5. Take note of the IP address displayed on the screen. You will need this address for the next step, activating the router. For example, the message below indicates that the router has IP address 10.42.0.162.

```
Congratulations! This device is now registered to info@paradrop.io.

The next step is to log into the device via ssh:

ssh paradrop@10.42.0.162
...
```

CHAPTER 7

# Developing Applications

This section of the document is devoted to describing the edge computing services (chutes) that run on ParaDrop. There are two categories of ParaDrop applications - pure edge applications and cloud-edge hybrid applications. The pure edge applications have standalone chutes, which can be deployed in the ParaDrop routers. Cloud-edge hybrid applications have both a cloud component and an edge component. In this section, we will focus on the chute development, in other words, the edge component.

## Introduction

ParaDrop is a software platform that enables services to run on Wi-Fi routers. We call these services *chutes* as in *parachutes*.

ParaDrop runs on top of Ubuntu Core, a lightweight, transactionally updated operating system designed for deployments on embedded and IoT devices, cloud and more. It runs a new breed of secure, remotely upgradeable Linux app packages known as snaps. We support chute deployment through containerization powered by Docker.

Minimally, a chute has a Dockerfile, which contains instructions for building and preparing the application to run on ParaDrop. A chute will usually also require scripts, binaries, configuration files, and other assets. For integration with the ParaDrop toolset, we highly recommend developing a chute as a GitHub project, but other organization methods are possible.

We will examine the hello-world chute as an example of a complete ParaDrop application.

### Structure

Our hello-world chute is a git project with the following files:

```
chute/index.html
Dockerfile
README.md
```

The top-level contains a README and a special file called "Dockerfile", which will be discussed below. As a convention, we place files that will be used by the running application in a subdirectory called "chute". This is not necessary but helps keep the project organized. Valid alternatives include "src" or "app".

### Dockerfile

The Dockerfile contains instructions for building and preparing an application to run on ParaDrop. Here is a minimal Dockerfile for our hello-world chute:

```
FROM nginx
ADD chute/index.html /usr/share/nginx/html/index.html
```

**FROM nginx**

The FROM instruction specifies a base image for the chute. This could be a Linux distribution such as "ubuntu:14.04" or an standalone application such as "nginx". The image name must match an image in the Docker public registry. We recommend choosing from the official repositories. Here we use "nginx" for a light-weight web server.

**ADD <source> <destination>**

The ADD instruction copies a file or directory from the source repository to the chute filesystem. This is useful for installing scripts or other files required by the chute and are part of the source repository. The <source> path should be inside the respository, and the <destination> path should be an absolute path or a path inside the chute's working directory. Here we install the index.html file from our source repository to the search directory used by nginx.

Other useful commands for building chutes are RUN and CMD. For a complete reference, please visit the official Dockerfile reference.

Here is an alternative implementation of the hello-world Dockerfile that demonstrates some of the other useful instructions.

```
FROM ubuntu:14.04
RUN apt-get update && apt-get install -y nginx
ADD chute/index.html /usr/share/nginx/html/index.html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Here we use a RUN instruction to install nginx and a CMD instruction to set nginx as the command to run inside the chute container. Using "ubuntu:14.04" as the base image gives access to any packages that can be installed through apt-get.

## Persistent Data

Each running chute has a persistent data storage that is not visible to other chutes. By default the persistent data directory is named "/data" inside the chute's filesystem. Files stored in this directory will remain when upgrading or downgrading the chute and are only removed when uninstalling the chute.

## System Information

The ParaDrop daemon share system information with chutes through a read-only directory named "/paradrop". Chutes that are configured with a WiFi access point will find a file in this directory that lists wireless clients. In future versions there will also be information about Bluetooth and other wireless devices.

### dnsmasq-wifi.leases

This file lists client devices that have connected to the chute's WiFi network and received a DHCP lease. This is a plain text file with one line for each device containing the following space-separated fields.

1. DHCP lease expiration time (seconds since Unix epoch).
2. MAC address.
3. IP address.
4. Host name, if known.

5. Client ID, if known; the format of this field varies between devices.

The following example shows two devices connected to the chute's WiFi network.

```
1480650200 00:11:22:33:44:55 192.168.128.130 android-ffeeddccbbaa9988 *
1480640500 00:22:44:66:88:aa 192.168.128.170 someones-iPod 01:00:22:44:66:88:aa
```

### Chute-to-Host API

The Paradrop daemon exposes some functionality and configuration options to running chutes through an HTTP API. This aspect of Paradrop is under rapid development, and new features will be added with every release. The host API is available to chutes through the URL "http://paradrop.io/api/v1". Paradrop automatically configures chutes to resolve "paradrop.io" to the ParaDrop device itself, so these requests go to the ParaDrop daemon running on the router and not to an outside server.

#### Authorization

In order to access the host API, chutes must pass a token with every request that proves the authenticity of the request. When chutes are installed on a ParaDrop router, they automatically receive a token through an environment variable named "PARADROP_API_TOKEN". The chute should read this environment variable and pass the token as a Bearer token in an HTTP Authorization header. Here is an example in Python using the Requests library.:

```
import os
import requests

CHUTE_NAME = os.environ.get('PARADROP_CHUTE_NAME', 'chute')
API_TOKEN = os.environ.get('PARADROP_API_TOKEN', 'NA')

headers = { 'Authorization': 'Bearer ' + API_TOKEN }
url = 'http://paradrop.io/api/v1/chutes/{}/networks'.format(CHUTE_NAME)
res = requests.get(url, headers=headers)
print(res.json())
```

Please refer to Host API Reference for a complete listing of API functions.

## Developing Light Chutes

Light chutes build and install the same way as normal chutes and can do many of the same things. However, they make use of prebuilt base images that are optimized for different programming languages.

Light chutes offer these advantages over normal chutes.

- **Safety:** Light chutes have stronger confinement properties, so you can feel safer installing a light chute written by a third party developer.
- **Fast installation:** Light chutes use a common base image that may already be cached on the router, so installation can be very fast.
- **Simplicity:** You do not need to learn how to write and debug a Dockerfile to develop a chute. Instead, you can use the package management tools you may already be using (e.g. package.json for npm and requirements.txt for pip).
- **Portability:** With ARM suppport coming soon for ParaDrop, your light chutes will most likely run on ARM with extra work on your part. This is not the case for normal chutes that use a custom Dockerfile.

We will look at the node-hello-world chute as an example of a light chute for ParaDrop.

### Structure

Our hello-world chute is a git project with the following files:

```
README.md
index.js
package.json
paradrop.yaml
```

The project contains the typical files for a node.js project as well as a special file called "paradrop.yaml".

### paradrop.yaml

The paradrop.yaml file contains information that ParaDrop needs in order to run the chute. Here are the contents for the hello-world example:

```
version: 1
type: light
use: node
command: node index.js
```

All of these fields are required and very simple to use.

**version: 1**

This specifies the version of the paradrop.yaml schema in order to allow future changes without breaking existing chutes. You should specify version *1*.

**type: light**

This indicates that we are building a *light* chute.

**use: node**

This indicates that we are using the *node* base image for this chute. You should choose the base image appropriate for your project. Supported images are: *node* and *python2*.

**command: node index.js**

This line indicates the command for starting your application. You can either specified it this way, as a string with spaces between the parameters, or you can use a list of strings. The latter format would be particularly useful if your parameters include spaces. Here is an example:

```
command:
  - node
  - index.js
```

### Persistent Data

Each running chute has a persistent data storage that is not visible to other chutes. By default the persistent data directory is named "/data" inside the chute's filesystem. Files stored in this directory will remain when upgrading or downgrading the chute and are only removed when uninstalling the chute.

## Getting Started with Go

This tutorial will teach you how to build a "Hello, World!" chute using Go.

## Prerequisites

Make sure you have Go installed as well as ParaDrop pdtools (v0.9.2 or newer).

```
pip install pdtools>=0.9.2
```

## Set up

Make a new directory.

```
mkdir go-hello-world
cd go-hello-world
```

## Create a chute configuration

Use the pdtools interactive init command to create a paradrop.yaml file for your chute.

```
python -m pdtools chute init
```

Use the following values as suggested responses to the prompts. If you have a different version of pdtools installed, the prompts may be slightly different.

```
name: go-hello-world
description: Hello World chute for ParaDrop using Go.
type: light
image: go
command: app
```

The end result should be a paradrop.yaml file similar to the following.

```
command: app
config: {}
description: Hello World chute for ParaDrop using Go.
name: go-hello-world
type: light
use: go
version: 1
```

## Develop the Application

Create a file name `main.go` with the following code.

```
package main

import (
    "fmt"
    "net/http"
)

func GetIndex(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello, World!\n")
}

func main() {
    fmt.Println("Listening on :8000")
```

```
    http.HandleFunc("/", GetIndex)
    http.ListenAndServe(":8000", nil)
}
```

Run the application locally with the following command.

```
go run main.go
```

Then load `http://localhost:8000/` in a web browser to see the result.

### Wrap Up

The web server in this application listens on port 8000. We need to include that information in the paradrop.yaml file as well. Use the following command to alter the configuration file.

```
python -m pdtools chute set config.web.port 8000
```

## Getting Started with Java

This tutorial will teach you how to build a "Hello, World!" chute using Java and Maven.

### Prerequisites

Make sure you have Java 1.8+, Maven 3.0+, as well as ParaDrop pdtools (v0.9.2 or newer).

```
pip install pdtools>=0.9.2
```

### Set up

Use Maven to set up an empty project.

```
mvn archetype:generate -DgroupId=org.paradrop.app -DartifactId=java-hello-world -DarchetypeArtifactId
cd java-hello-world
```

### Create a chute configuration

Use the pdtools interactive init command to create a paradrop.yaml file for your chute.

```
python -m pdtools chute init
```

Use the following values as suggested responses to the prompts. If you have a different version of pdtools installed, the prompts may be slightly different.

```
name: java-hello-world
description: Hello World chute for ParaDrop using Java.
type: light
image: maven
command: java -cp target/java-hello-world-1.0-SNAPSHOT.jar org.paradrop.app.App
```

The end result should be a paradrop.yaml file similar to the following.

```
command: java -cp target/java-hello-world-1.0-SNAPSHOT.jar org.paradrop.app.App
config: {}
description: Hello World chute for ParaDrop using Java.
name: java-hello-world
type: light
use: maven
version: 1
```

## Develop the Application

Replace the automatically-generated application code in `src/main/java/org/paradrop/app/App.java` with the following code.

```
package org.paradrop.app;

import java.io.IOException;
import java.io.OutputStream;
import java.net.InetSocketAddress;

import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpHandler;
import com.sun.net.httpserver.HttpServer;

public class App {
    public static void main(String[] args) throws Exception {
        System.out.println("Listening on :8000");
        HttpServer server = HttpServer.create(new InetSocketAddress(8000), 0);
        server.createContext("/", new GetIndex());
        server.start();
    }

    static class GetIndex implements HttpHandler {
        @Override
        public void handle(HttpExchange t) throws IOException {
            String response = "Hello, World!";
            t.sendResponseHeaders(200, response.length());
            OutputStream os = t.getResponseBody();
            os.write(response.getBytes());
            os.close();
        }
    }
}
```

Run the application locally with the following commands.

```
mvn package
java -cp target/java-hello-world-1.0-SNAPSHOT.jar org.paradrop.app.App
```

Then load `http://localhost:8000/` in a web browser to see the result.

## Wrap Up

The web server in this application listens on port 8000. We need to include that information in the paradrop.yaml file as well. Use the following command to alter the configuration file.

```
python -m pdtools chute set config.web.port 8000
```

# Getting Started with Node.js

This tutorial will teach you how to build a "Hello, World!" chute using Node.js and Express.

## Prerequisites

Make sure you have Node.js (v6 or newer) installed as well as ParaDrop pdtools (v0.9.2 or newer).

```
pip install pdtools>=0.9.2
```

## Set up

Make a new directory.

```
mkdir node-hello-world
cd node-hello-world
```

## Create a chute configuration

Use the pdtools interactive init command to create a paradrop.yaml file for your chute.

```
python -m pdtools chute init
```

Use the following values as suggested responses to the prompts. If you have a different version of pdtools installed, the prompts may be slightly different.

```
name: node-hello-world
description: Hello World chute for ParaDrop using Node.js.
type: light
image: node
command: node index.js
```

The end result should be a paradrop.yaml file similar to the following.

```
command: node index.js
config: {}
description: Hello World chute for ParaDrop using Node.js.
name: node-hello-world
type: light
use: node
version: 1
```

The `pdtools chute init` command will also create a package.json file for you if one did not already exist, so there is no need to run `npm init` after running `pdtools chute init`.

## Install Dependencies

Use the following command to install some dependencies. We will be using Express as a simple web server.

The `--save` option instructs npm to save the packages to the package.json file. When installing the chute, ParaDrop will read package.json to install the same versions of the packages that you used for development.:

```
npm install --save express@^4.16.1
```

### Develop the Application

We indicated that index.js is the entrypoint for the application, so we will create a file named `index.js` and put our code there.

```
const express = require('express')
const app = express()

app.get('/', function (req, res) {
  res.send('Hello, World!')
})

app.listen(3000, function() {
  console.log('Listening on port 3000.')
})
```

Run the application locally with the following command.

```
node index.js
```

Then load `http://localhost:3000/` in a web browser to see the result.

### Wrap Up

The web server in this application listens on port 3000. We need to include that information in the paradrop.yaml file as well. Use the following command to alter the configuration file.

```
python -m pdtools chute set config.web.port 3000
```

## Getting Started with Python

This tutorial will teach you how to build a "Hello, World!" chute using Python and Flask.

### Prerequisites

Make sure you have Python 2 installed as well as ParaDrop pdtools (v0.9.2 or newer).

```
pip install pdtools>=0.9.2
```

### Set up

Make a new directory.

```
mkdir python-hello-world
cd python-hello-world
```

## Create a chute configuration

Use the pdtools interactive init command to create a paradrop.yaml file for your chute.

```
python -m pdtools chute init
```

Use the following values as suggested responses to the prompts. If you have a different version of pdtools installed, the prompts may be slightly different.

```
name: python-hello-world
description: Hello World chute for ParaDrop using Python.
type: light
image: python2
command: python2 -u main.py
```

The end result should be a paradrop.yaml file similar to the following.

```
command: python2 -u main.py
config: {}
description: Hello World chute for ParaDrop using Python.
name: python-hello-world
type: light
use: python2
version: 1
```

## Install Dependencies

We will use pip and virtualenv to manage dependencies for the project. First set up a virtual enviroment.

```
virtualenv venv
source venv/bin/activate
```

Use the following command to install some dependencies. We will be using Flask as a simple web server.

```
pip install Flask==0.12.2
```

Finally, save the version information to a file called `requirements.txt`. When installing the chute, ParaDrop will use this file to install the same versions of the packages that you used during development.

```
pip freeze >requirements.txt
```

## Develop the Application

We indicated that main.py is the entrypoint for the application, so we will create a file named `main.py` and put our code there.

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def index():
    return 'Hello, World!'

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

Run the application locally with the following command.

```
python main.py
```

Then load `http://localhost:5000/` in a web browser to see the result.

### Wrap Up

The web server in this application listens on port 5000. We need to include that information in the paradrop.yaml file as well. Use the following command to alter the configuration file.

```
python -m pdtools chute set config.web.port 5000
```

## Tutorial: Sticky Board

This tutorial will teach you how to build a fully-functional ParaDrop application from scratch. Through the tutorial, we will build a "Sticky Board", a local board where visitors can post images for others to see. We will be using Node.js to build the application, so make sure you have that installed on your development machine.

### Set Up

Make a new directory, and initialize a git repository:

```
mkdir sticky_board
cd sticky_board
git init
mkdir views
```

### Setup Node.js Project

We will be using npm to manage Node.js packages. You can use the `npm init` command to get started or create a file called package.json. with the following contents:

```
{
  "name": "sticky_board",
  "version": "1.0.0",
  "description": "Post images for others to see.",
  "main": "index.js",
  "author": "ParaDrop Team"
}
```

### Install Dependencies

Use the following command to install some dependencies that we will be using to build the application. We use express as a simple web server along with a plugin for accepting file uploads. We will also use Embedded JS (EJS) for simple templating, demonstrated later in this tutorial.

The `--save` option instructs npm to save the packages to the package.json file. ParaDrop will read package.json to install the same versions of the packages that you used for development.

```
npm install --save ejs@^2.5.6 express@^4.14.1 express-fileupload@^0.1.1
```

## Hello World

Let's start with a minimal Hello World Express.js example. Create a file named `index.js` and add the following code:

```
var express = require('express');
var app = express();

app.get('/', function (req, res) {
  res.send('Hello World!');
});

app.listen(3000, function() {
  console.log('Listening on port 3000.');
});
```

Run the app with the following command:

```
node index.js
```

Then load `http://localhost:3000/` in a web browser to see the result.

## Image Uploads

Next, we will add an endpoint to receive image uploads.

```
var express = require('express');
var fileupload = require('express-fileupload');

var app = express();

// Use PARADROP_DATA_DIR when running on Paradrop and /tmp for testing.
var storage_dir = process.env.PARADROP_DATA_DIR || '/tmp';

app.use(fileupload());
app.use(express.static(storage_dir));
app.set('view engine', 'ejs');

app.post('/create', function(req, res) {
  var img = req.files.img;
  if (img) {
    img.mv(storage_dir + '/' + img.name);
  }

  res.redirect('/');
});

app.get('/', function (req, res) {
  res.render('home');
});

app.listen(3000, function() {
  console.log('Listening on port 3000.');
});
```

Create a new file in the views directory called home.ejs with the following contents:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>ParaDrop Sticky Board</title>
  </head>
  <body>
    <h1>ParaDrop Sticky Board</h1>
    <h2>Create a Note</h2>
    <p>Upload an image file to create a note for others to see.</p>
    <form action="/create" method="POST" encType="multipart/form-data">
      <input type="file" name="img" />
      <input type="submit" value="Create" />
    </form>
  </body>
</html>
```

Right now it is just plain HTML. In the next section we will make use of templating to add images to the sticky board.

Run the app again and load `http://localhost:3000/`. Try using the form to upload an image. You should then be able to find your image by loading `http://localhost:3000/<filename>`.

## Displaying Notes

The last thing the app needs to be able to do is display all of the notes that people have posted. First, add some logic to index.js to keep track of the most recent image uploads:

```
var express = require('express');
var fileupload = require('express-fileupload');

var app = express();

// Use PARADROP_DATA_DIR when running on Paradrop and /tmp for testing.
var storage_dir = process.env.PARADROP_DATA_DIR || '/tmp';

// Maximum number of notes to display.
var max_visible_notes = process.env.MAX_VISIBLE_NOTES || 16;

app.locals.notes = [];
for (var i = 0; i < max_visible_notes; i++) {
  if (i % 2 == 0) {
    addNote('http://pages.cs.wisc.edu/~hartung/paradrop/paradrop.png');
  } else {
    addNote('http://pages.cs.wisc.edu/~hartung/paradrop/paradrop_inverted.png');
  }
}

function addNote(img) {
  app.locals.notes.push({
    img: img,
  });

  if (app.locals.notes.length > max_visible_notes) {
    app.locals.notes = app.locals.notes.slice(-max_visible_notes);
  }
}
```

```
app.use(fileupload());
app.use(express.static(storage_dir));
app.set('view engine', 'ejs');

app.post('/create', function(req, res) {
  var img = req.files.img;
  if (img) {
    img.mv(storage_dir + '/' + img.name);
    addNote(img.name);
  }

  res.redirect('/');
});

app.get('/', function (req, res) {
  res.render('home');
});

app.listen(3000, function() {
  console.log('Listening on port 3000.');
});
```

The paradrop.png and paradrop_inverted.png are just used as fillers until people post other images. Feel free to use different images.

Also, update home.ejs:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>ParaDrop Sticky Board</title>

    <style>
      div.holder {
        float: left;
        min-width: 240px;
        width: 24%;
        padding: 5px 5px;
      }

      div.separator {
        clear: both;
      }
    </style>
  </head>
  <body>
    <h1>ParaDrop Sticky Board</h1>

    <div>
      <% for(var i = 0; i<notes.length; i++) {%>
        <div class="holder">
          <img src="<%= notes[i].img %>" width="100%"></img>
        </div>
      <% } %>
    </div>

    <div class="separator"></div>
```

```
    <h2>Create a Note</h2>
    <p>Upload an image file to create a note for others to see.</p>
    <form action="/create" method="POST" encType="multipart/form-data">
      <input type="file" name="img" />
      <input type="submit" value="Create" />
    </form>
  </body>
</html>
```

We use some Embedded JS code to loop over the array of notes stored in `app.locals.notes` and generate an img element for each one with the appropriate filename.

Now when you run the app and load `http://localhost:3000/` you should see the filler images. Try using the form to upload an image, and it should appear on the board.

## Preparing the Chute

Create a file called paradrop.yaml with the following contents:

```
version: 1
type: light
use: node
command: node index.js
```

This file tells ParaDrop a few things about how to run your code on a ParaDrop gateway.

Finally, add all of your new files to the git repository:

```
git add index.js package.json paradrop.yaml views/home.ejs
git commit -m "Created sticky board from tutorial"
```

Create a new repository on github.com and follow their instructions to push your code to github.

## Registering the Chute with ParaDrop

Log on to paradrop.org and go to the Chute Store tab. Click "Create Chute" and give your chute a name and description. You may need to be creative with the name because the chute store requires unique names. Then click "Submit".

Next, click "Create Version". For this tutorial, there are only two important fields to fill out on this form. First, check the box to "enable web service" and enter the number 3000 because that is the port we chose in index.js. Second, select "Download from URL" for Project source and enter the github URL for your project. Then click "Submit".

Congratulations! You have made a ParaDrop chute. If you have a ParaDrop router, you should now be able to install the chute on your router. If not, you can follow the Getting Started guide to set up a VM running ParaDrop.

CHAPTER 8

# Frequently Asked Questions

Please check here for issues or questions that commonly arise.

## Issues with the hardware or operating system

### Issue 1: Docker fails to start after a reboot

This can happen if either the *docker.pid* file or the *docker-containerd.pid* file was not properly cleaned up on system reboot, which causes the Docker daemon to conclude that it is already running.

To fix this, remove the pid file on the router and reboot.

```
sudo rm /var/snap/docker/current/run/docker.pid
sudo rm /var/snap/docker/current/run/docker/libcontainerd/docker-containerd.pid
sudo reboot
```

Occasionally Docker will crash and not restart properly even after a reboot. We find that disabling and re-enabling the service helps in such cases.

```
sudo snap disable docker
sudo snap enable docker
```

### Issue 2: WiFi devices are not detected after a reboot

Occasionally, when routers start up the WiFi devices are not detected properly. When this happens the command `iw dev` will display nothing instead of the expected devices. This is usually remedied by rebooting. A global setting to reboot the router if WiFi devices are missing is available on the router settings page.

CHAPTER 9

# How to Contribute

This section focuses on the *ParaDrop Daemon*. This is the set of daemons and tools required to allow the Paradrop platform to function on virtual machines and real hardware. ParaDrop is an open source project. The source code of the ParaDrop daemon is available at github. Issue reports and pull requests are welcomed.

## ParaDrop daemon development

ParaDrop repository includes a set of tools to make development as easy as possible.

Currently this system takes the form of a bash script that automates installation and execution. This page outlines the steps required to manually install the dependencies, build the package and install the package into hardware/VMs.

We recommend using Ubuntu 16.04 LTS as the development environment for this version of ParaDrop because we use snapcraft to package and distribute the ParaDrop daemon.

You will only need to follow these instructions if you will be making changes to the ParaDrop daemon. Otherwise, you can use our pre-built ParaDrop snap or disk image from ParaDrop release.

### Building ParaDrop daemon

pdbuild.sh is the script we work with during the development. It provides following commands:

- `./pdbuild.sh setup` installs development dependencies.
- `./pdbuild.sh run` executes the ParaDrop daemon locally in the development machine. It is useful for debugging.
- `./pdbuild.sh build` builds the snap package. Check snapcraft documentation for detailed information about snap packages and snapcraft.
- `./pdbuild.sh image` builds the ubuntu core image that we can flash into SD card or SSD module of a ParaDrop router. It pre-installs the required snaps for us automatically, e.g. docker.

### Installing ParaDrop into hardware/VMs

After the ParaDrop daemon snap is ready (paradrop-daemon_<version>_amd64.snap), we can install it on a ParaDrop router. Check Hardware Support for information about preparing a ParaDrop router.

Copy the paradrop snap to the router with ParaDrop image installed:

```
scp paradrop-daemon-<version>_amd64.snap paradrop@<router ip>:
```

Then we can log in to a ParaDrop router:

```
ssh paradrop@<router ip>
```

Install the dependent snaps in a ParaDrop router:

```
snap install docker
```

Install the newly created ParaDrop daemon snap package:

```
snap install --devmode paradrop-daemon-<version>_amd64.snap
```

### Checking logs of ParaDrop daemon

After install the ParaDrop daemon, we can use 'pdlog' to check the log of ParaDrop daemon on the ParaDrop router:

```
paradrop-daemon.pdlog -f
```

### Building ParaDrop tools

We have published the ParaDrop tools snap in the Ubuntu Snap Store. On the development machine, we can install it with below command:

```
snap install paradrop-tools
```

Get the manual of ParaDrop tools:

```
paradrop-tools.pdtools --help
```

More detailed information about ParaDrop tools can be find in Developing Applications. The git repository of ParaDrop includes the source code of ParaDrop tools. Developers can build the latest version of ParaDrop tools by running below command in the folder 'tools':

```
snapcraft
```

## Documentation and tests

Documentation is handled by sphinx and readthedocs.

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

### Documentation

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 `paradrop/daemon/paradrop/` and build documentation from the docstrings within.

For example, the directive `.. automodule:: paradrop.backend` will build all the documentation for the given package. See Docstring Conventions for details.

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/paradrop paradrop/daemon/paradrop/
```

This scans packages in the `paradrop/daemon/paradrop` directory and creates .rst files in `docs/paradrop`.

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.

## 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
```

Install the required packages:

```
pip install -r docs/requirements.txt
```

Run all tests:

```
nosetests
```

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 10

# Host API Reference

## Chute Management

Install and manage chutes on the host.

Endpoints for these functions can be found under /api/v1/chutes.

**GET /api/v1/chutes/**
List installed chutes.

**Example request**:

```
GET /api/v1/chutes/
```

**Example response**:

```
HTTP/1.1 200 OK
Content-Type: application/json

[
  {
    "environment": {},
    "name": "hello-world",
    "allocation": {
      "cpu_shares": 1024,
      "prioritize_traffic": false
    },
    "state": "running",
    "version": "x1511808778",
    "resources": null
  }
]
```

**GET /api/v1/chutes/**(*chute*)**/networks/***network***/stations/***mac** Get detailed information about a connected station.

**Example request**:

```
GET /api/v1/chutes/captive-portal/networks/wifi/stations/5c:59:48:7d:b9:e6
```

**Example response**:

```
HTTP/1.1 200 OK
Content-Type: application/json
```

```
{
  "rx_packets": "230",
  "tdls_peer": "no",
  "authenticated": "yes",
  "rx_bytes": "12511",
  "tx_bitrate": "1.0 MBit/s",
  "tx_retries": "0",
  "signal": "-45 [-49, -48] dBm",
  "authorized": "yes",
  "rx_bitrate": "65.0 MBit/s MCS 7",
  "mfp": "no",
  "tx_failed": "0",
  "inactive_time": "4688 ms",
  "mac_addr": "5c:59:48:7d:b9:e6",
  "tx_bytes": "34176",
  "wmm_wme": "yes",
  "preamble": "short",
  "tx_packets": "88",
  "signal_avg": "-44 [-48, -47] dBm"
}
```

**GET** **/api/v1/chutes/** (*chute*) **/networks/** *network***/hostapd_status** Get low-level status information from the access point.

**Example request**:

```
GET /api/v1/chutes/captive-portal/networks/wifi/hostapd_status
```

**Example response**:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "olbc_ht": "0",
  "cac_time_left_seconds": "N/A",
  "num_sta_no_short_slot_time": "0",
  "olbc": "1",
  "num_sta_non_erp": "0",
  "ht_op_mode": "0x4",
  "state": "ENABLED",
  "num_sta_ht40_intolerant": "0",
  "channel": "11",
  "bssid[0]": "02:00:08:24:03:dd",
  "ieee80211n": "1",
  "cac_time_seconds": "0",
  "num_sta[0]": "1",
  "ieee80211ac": "0",
  "phy": "phy0",
  "num_sta_ht_no_gf": "1",
  "freq": "2462",
  "num_sta_ht_20_mhz": "1",
  "num_sta_no_short_preamble": "0",
  "secondary_channel": "0",
  "ssid[0]": "Free WiFi",
  "num_sta_no_ht": "0",
  "bss[0]": "vwlan7e1b"
}
```

**GET /api/v1/chutes/**(*chute*)**/networks/***network*/**stations** Get detailed information about connected wireless stations.

**Example request**:

```
GET /api/v1/chutes/captive-portal/networks/wifi/stations
```

**Example response**:

```
HTTP/1.1 200 OK
Content-Type: application/json

[
  {
    "rx_packets": "230",
    "tdls_peer": "no",
    "authenticated": "yes",
    "rx_bytes": "12511",
    "tx_bitrate": "1.0 MBit/s",
    "tx_retries": "0",
    "signal": "-45 [-49, -48] dBm",
    "authorized": "yes",
    "rx_bitrate": "65.0 MBit/s MCS 7",
    "mfp": "no",
    "tx_failed": "0",
    "inactive_time": "4688 ms",
    "mac_addr": "5c:59:48:7d:b9:e6",
    "tx_bytes": "34176",
    "wmm_wme": "yes",
    "preamble": "short",
    "tx_packets": "88",
    "signal_avg": "-44 [-48, -47] dBm"
  }
]
```

**GET /api/v1/chutes/**(*chute*)**/networks/***network*/**leases** Get current list of DHCP leases for chute network.

Returns a list of DHCP lease records with the following fields:

**expires** lease expiration time (seconds since Unix epoch)

**mac_addr** device MAC address

**ip_addr** device IP address

**hostname** name that the device reported

**client_id** optional identifier supplied by device

**Example request**:

```
GET /api/v1/chutes/captive-portal/networks/wifi/leases
```

**Example response**:

```
HTTP/1.1 200 OK
Content-Type: application/json

[
  {
    "client_id": "01:5c:59:48:7d:b9:e6",
```

```
        "expires": "1511816276",
        "ip_addr": "192.168.128.64",
        "mac_addr": "5c:59:48:7d:b9:e6",
        "hostname": "paradrops-iPod"
      }
    ]
```

**GET /api/v1/chutes/** (*chute*) **/networks/***network***/ssid** Get currently configured SSID for the chute network.

**Example request**:

```
GET /api/v1/chutes/captive-portal/networks/wifi/ssid
```

**Example response**:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "ssid": "Free WiFi",
  "bssid": "02:00:08:24:03:dd"
}
```

**PUT /api/v1/chutes/** (*chute*) **/networks/***network***/ssid** Change the configured SSID for the chute network.

The change will not persist after a reboot. If a persistent change is desired, you should update the chute configuration instead.

**Example request**:

```
PUT /api/v1/chutes/captive-portal/networks/wifi/ssid
Content-Type: application/json

{
  "ssid": "Best Free WiFi"
}
```

**Example response**:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "message": "OK"
}
```

**GET /api/v1/chutes/** (*chute*) **/networks/***network* Get information about a network configured for the chute.

**Example request**:

```
GET /api/v1/chutes/captive-portal/networks/wifi
```

**Example response**:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
```

```
    "interface": "wlan0",
    "type": "wifi",
    "name": "wifi"
  }
```

**GET /api/v1/chutes/**(*chute*)**/networks**
Get list of networks configured for the chute.

**Example request**:

```
GET /api/v1/chutes/captive-portal/networks
```

**Example response**:

```
HTTP/1.1 200 OK
Content-Type: application/json

[
  {
    "interface": "wlan0",
    "type": "wifi",
    "name": "wifi"
  }
]
```

**GET /api/v1/chutes/**(*chute*)**/config**
Get current chute configuration.

**Example request**:

```
GET /api/v1/chutes/captive-portal/config
```

**Example response**:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "net": {
    "wifi": {
      "dhcp": {
        "lease": "1h",
        "limit": 250,
        "start": 3
      },
      "intfName": "wlan0",
      "options": {
        "isolate": True
      },
      "ssid": "Free WiFi",
      "type": "wifi"
    }
  }
}
```

**PUT /api/v1/chutes/**(*chute*)**/config**
Update the chute configuration and restart to apply changes.

**Example request**:

```
PUT /api/v1/chutes/captive-portal/config
Content-Type: application/json

{
  "net": {
    "wifi": {
      "dhcp": {
        "lease": "1h",
        "limit": 250,
        "start": 3
      },
      "intfName": "wlan0",
      "options": {
        "isolate": True
      },
      "ssid": "Better Free WiFi",
      "type": "wifi"
    }
  }
}
```

**Example response**:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "change_id": 1
}
```

**GET /api/v1/chutes/**(*chute*)**/cache**
Get chute cache contents.

The chute cache is a key-value store used during chute installation. It can be useful for debugging the Paradrop platform.

**GET /api/v1/chutes/**(*chute*)
Get information about an installed chute.

**Example request**:

```
GET /api/v1/chutes/hello-world
```

**Example response**:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "environment": {},
  "name": "hello-world",
  "allocation": {
    "cpu_shares": 1024,
    "prioritize_traffic": false
  },
  "state": "running",
  "version": "x1511808778",
  "resources": null
}
```

## Device Configuration

This module exposes device configuration.

Endpoints for these functions can be found under /api/v1/config.

**POST /api/v1/config/factoryReset**
Initiate the factory reset process.

**PUT /api/v1/config/hostconfig**
Replace the device's host configuration.

**Example request**:

```
PUT /api/v1/config/hostconfig
Content-Type: application/json

{
  "firewall": {
    "defaults": {
      "forward": "ACCEPT",
      "input": "ACCEPT",
      "output": "ACCEPT"
    }
  },
  ...
}
```

**Example response**:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  change_id: 1
}
```

For a complete example, please see the Host Configuration section.

**GET /api/v1/config/hostconfig**
Get the device's current host configuration.

**Example request**:

```
GET /api/v1/config/hostconfig
```

**Example response**:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "firewall": {
    "defaults": {
      "forward": "ACCEPT",
      "input": "ACCEPT",
      "output": "ACCEPT"
    }
  },
  ...
}
```

For a complete example, please see the Host Configuration section.

**POST /api/v1/config/provision**
Provision the device with credentials from a cloud controller.

**GET /api/v1/config/provision**
Get the provision status of the device.

**GET /api/v1/config/pdconf**
Get configuration sections from pdconf.

This returns a list of configuration sections and whether they were successfully applied. This is intended for debugging purposes.

**PUT /api/v1/config/pdconf**
Trigger pdconf to reload UCI configuration files.

Trigger pdconf to reload UCI configuration files and return the status. This function is intended for low-level debugging of the paradrop pdconf module.

**GET /api/v1/config/pdid**
Get the device's current ParaDrop ID. This is the identifier assigned by the cloud controller.

**Example request**:

```
GET /api/v1/config/pdid
```

**Example response**:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  pdid: "5890e1e5ab7e317e6c6e049f"
}
```

**POST /api/v1/config/sshKeys/**(*user*)
Manage list of authorized keys for SSH access.

**GET /api/v1/config/sshKeys/**(*user*)
Manage list of authorized keys for SSH access.

## Device Information

Provide information of the router, e.g. board version, CPU information, memory size, disk size.

Endpoints for these functions can be found under /api/v1/info.

**GET /api/v1/info/environment**
Get environment variables.

Returns a dictionary containing the environment variables passed to the Paradrop daemon. This is useful for development and debugging purposes (e.g. see how PATH is set on Paradrop when running in different contexts).

**Example request**:

```
GET /api/v1/info/environment
```

**Example response**:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "LANG": "C.UTF-8",
  "SNAP_REVISION": "x73",
  "SNAP_COMMON": "/var/snap/paradrop-daemon/common",
  "XDG_RUNTIME_DIR": "/run/user/0/snap.paradrop-daemon",
  "SNAP_USER_COMMON": "/root/snap/paradrop-daemon/common",
  "SNAP_LIBRARY_PATH": "/var/lib/snapd/lib/gl:/var/lib/snapd/void",
  "SNAP_NAME": "paradrop-daemon",
  "PWD": "/var/snap/paradrop-daemon/x73",
  "PATH": "/snap/paradrop-daemon/x73/usr/sbin:/snap/paradrop-daemon/x73/usr/bin:/snap/paradrop-d
  "SNAP": "/snap/paradrop-daemon/x73",
  "SNAP_DATA": "/var/snap/paradrop-daemon/x73",
  "SNAP_VERSION": "0.9.2",
  "SNAP_ARCH": "amd64",
  "SNAP_USER_DATA": "/root/snap/paradrop-daemon/x73",
  "TEMPDIR": "/tmp",
  "HOME": "/root/snap/paradrop-daemon/x73",
  "SNAP_REEXEC": "",
  "LD_LIBRARY_PATH": "/var/lib/snapd/lib/gl:/var/lib/snapd/void:/snap/paradrop-daemon/x73/usr/li
  "TMPDIR": "/tmp"
  ...
}
```

**GET /api/v1/info/telemetry**
Get a telemetry report.

This contains information about resource utilization by chute and system totals. This endpoint returns the same data that we periodically send to the controller if telemetry is enabled.

**GET /api/v1/info/hardware**
Get information about the hardware platform.

**Example request**:

```
GET /api/v1/info/hardware
```

**Example response**:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "wifi": [
    {
      "slot": "pci/0000:04:00.0",
      "vendorId": "0x168c",
      "macAddr": "04:f0:21:2f:b7:c1",
      "id": "pci-wifi-0",
      "deviceId": "0x003c"
    },
    {
      "slot": "pci/0000:06:00.0",
      "vendorId": "0x168c",
      "macAddr": "04:f0:21:0f:78:28",
      "id": "pci-wifi-1",
      "deviceId": "0x002a"
    }
```

```
    ],
    "memory": 2065195008,
    "vendor": "PC Engines",
    "board": "APU 1.0",
    "cpu": "x86_64"
}
```

**GET /api/v1/info/software**
Get information about the operating system.

Returns a dictionary containing information the BIOS version, OS version, kernel version, Paradrop version, and system uptime.

**Example request**:

```
GET /api/v1/info/software
```

**Example response**:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "biosVersion": "SageBios_PCEngines_APU-45",
  "biosDate": "04/05/2014",
  "uptime": 15351,
  "kernelVersion": "Linux-4.4.0-101-generic",
  "pdVersion": "0.9.2",
  "biosVendor": "coreboot",
  "osVersion": "Ubuntu 4.4.0-101.124-generic 4.4.95"
}
```

**GET /api/v1/info/features**
Get features supported by the host.

This is a list of strings specifying features supported by the daemon.

**Explanation of feature strings**:

**hostapd-control** The daemon supports the hostapd control interface and provides a websocket channel for accessing it.

**Example request**:

```
GET /api/v1/info/features
```

**Example response**:

```
HTTP/1.1 200 OK
Content-Type: application/json

[
  "hostapd-control"
]
```

CHAPTER 11

# Source Code Reference

## Subpackages

### paradrop.airshark package

#### Submodules

#### paradrop.airshark.airshark module

**class** **AirsharkManager**
Bases: object

**add_analyzer_observer**(*observer*)

**add_spectrum_observer**(*observer*)

**check_spectrum**()

**on_analyzer_message**(*message*)

**on_interface_down**(*interface*)

**on_interface_up**(*interface*)

**read_raw_samples**()

**remove_analyzer_observer**(*observer*)

**remove_spectrum_observer**(*observer*)

**status**()

#### paradrop.airshark.analyzer module

**class** **AnalyzerProcessProtocol**(*airshark_manager*)
Bases: twisted.internet.protocol.ProcessProtocol

**childDataReceived**(*childFd*, *data*)

**connectionMade**()

**feedSpectrumData**(*data*)

**isRunning**()

**processEnded**(*status*)

**stop**()

### paradrop.airshark.scanner module

**class Scanner**(*interface*)
Bases: object

**cmd_chanscan**()

**cmd_disable**()

**cmd_set_samplecount**(*count*)

**cmd_set_short_repeat**(*short_repeat*)

**debugfs_dir = None**

**dev_to_phy**(*dev*)

**freqlist = None**

**get_debugfs_dir**()

**interface = None**

**process = None**

**set_freqs**(*minf*, *maxf*, *spacing*)

**spectrum_reader = None**

**start**()

**stop**()

### paradrop.airshark.spectrum_reader module

**class SpectrumReader**(*path*)
Bases: object

**static decode**(*data*)
For information about the decoding of spectral samples see: https://wireless.wiki.kernel.org/en/users/drivers/ath9k/spectral_s
https://github.com/erikarn/ath_radar_stuff/tree/master/lib and your ath9k implementation in e.g. /drivers/net/wireless/ath/ath9k/common-spectral.c

**flush**()

**hdrsize = 3**

**pktsize = 73**

**read_samples**()

**sc_wide = 0.3125**

### Module contents

## paradrop.backend package

### Submodules

### paradrop.backend.airshark_api module

APIs for developers to check whether Airshark feature is available or not

**class AirsharkApi** (*airshark_manager*)

**routes**
L{Klein} is an object which is responsible for maintaining the routing configuration of our application.

**@ivar _url_map: A C{werkzeug.routing.Map} object which will be used for** routing resolution.

@ivar _endpoints: A C{dict} mapping endpoint names to handler functions.

**status** (*request*)

### paradrop.backend.airshark_ws module

**class AirsharkAnalyzerFactory** (*airshark_manager*, **args*, ***kwargs*)
Bases: autobahn.twisted.websocket.WebSocketServerFactory

**buildProtocol** (*addr*)

**class AirsharkAnalyzerProtocol** (*factory*)
Bases: autobahn.twisted.websocket.WebSocketServerProtocol

**onClose** (*wasClean*, *code*, *reason*)

**onOpen** ()

**on_analyzer_message** (*message*)

**class AirsharkSpectrumFactory** (*airshark_manager*, **args*, ***kwargs*)
Bases: autobahn.twisted.websocket.WebSocketServerFactory

**buildProtocol** (*addr*)

**class AirsharkSpectrumProtocol** (*factory*)
Bases: autobahn.twisted.websocket.WebSocketServerProtocol

**onClose** (*wasClean*, *code*, *reason*)

**onOpen** ()

**on_spectrum_data** (*data*)

### paradrop.backend.auth module

**class AuthApi** (*password_manager*, *token_manager*)
Bases: object

**local_login** (*request*)
Login using local authentication (username+password).

**routes**
L{Klein} is an object which is responsible for maintaining the routing configuration of our application.

**@ivar _url_map: A C{werkzeug.routing.Map} object which will be used for** routing resolution.

@ivar _endpoints: A C{dict} mapping endpoint names to handler functions.

**check_auth** (*password_manager*, *token_manager*, *auth_header*)

**get_allowed_bearer**()
Return set of allowed bearer tokens.

**get_username_password**(*userpass*)
Please note: username and password can either be presented in plain text such as "admin:password" or base64 encoded such as "YWRtaW46cGFzc3dvcmQ=". Both forms should be returned from this function.

**requires_auth**(*func*)
Use as a decorator for API functions to require authorization.

This checks the Authorization HTTP header. It handles username and password as well as bearer tokens.

**verify_password**(*password_manager*, *userpass*)

### paradrop.backend.chute_api module

Install and manage chutes on the host.

Endpoints for these functions can be found under /api/v1/chutes.

**class ChuteApi**(*update_manager*)
Bases: object

**create_chute**(*request*)

**delete_chute**(*request*, *chute*)

**delete_station**(*request*, *chute*, *network*, *mac*)

**get_chute**(*request*, *chute*)
Get information about an installed chute.

**Example request**:

```
GET /api/v1/chutes/hello-world
```

**Example response**:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "environment": {},
  "name": "hello-world",
  "allocation": {
    "cpu_shares": 1024,
    "prioritize_traffic": false
  },
  "state": "running",
  "version": "x1511808778",
  "resources": null
}
```

**get_chute_cache**(*request*, *chute*)
Get chute cache contents.

The chute cache is a key-value store used during chute installation. It can be useful for debugging the Paradrop platform.

**get_chute_config**(*request*, *chute*)
Get current chute configuration.

**Example request**:

```
GET /api/v1/chutes/captive-portal/config
```

**Example response**:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "net": {
    "wifi": {
      "dhcp": {
        "lease": "1h",
        "limit": 250,
        "start": 3
      },
      "intfName": "wlan0",
      "options": {
        "isolate": True
      },
      "ssid": "Free WiFi",
      "type": "wifi"
    }
  }
}
```

**get_chutes** (*request*)
List installed chutes.

**Example request**:

```
GET /api/v1/chutes/
```

**Example response**:

```
HTTP/1.1 200 OK
Content-Type: application/json

[
  {
    "environment": {},
    "name": "hello-world",
    "allocation": {
      "cpu_shares": 1024,
      "prioritize_traffic": false
    },
    "state": "running",
    "version": "x1511808778",
    "resources": null
  }
]
```

**get_hostapd_status** (*request*, *chute*, *network*)
Get low-level status information from the access point.

**Example request**:

```
GET /api/v1/chutes/captive-portal/networks/wifi/hostapd_status
```

**Example response**:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "olbc_ht": "0",
  "cac_time_left_seconds": "N/A",
  "num_sta_no_short_slot_time": "0",
  "olbc": "1",
  "num_sta_non_erp": "0",
  "ht_op_mode": "0x4",
  "state": "ENABLED",
  "num_sta_ht40_intolerant": "0",
  "channel": "11",
  "bssid[0]": "02:00:08:24:03:dd",
  "ieee80211n": "1",
  "cac_time_seconds": "0",
  "num_sta[0]": "1",
  "ieee80211ac": "0",
  "phy": "phy0",
  "num_sta_ht_no_gf": "1",
  "freq": "2462",
  "num_sta_ht_20_mhz": "1",
  "num_sta_no_short_preamble": "0",
  "secondary_channel": "0",
  "ssid[0]": "Free WiFi",
  "num_sta_no_ht": "0",
  "bss[0]": "vwlan7e1b"
}
```

**get_leases** (*request*, *chute*, *network*)

Get current list of DHCP leases for chute network.

Returns a list of DHCP lease records with the following fields:

**expires** lease expiration time (seconds since Unix epoch)

**mac_addr** device MAC address

**ip_addr** device IP address

**hostname** name that the device reported

**client_id** optional identifier supplied by device

**Example request**:

```
GET /api/v1/chutes/captive-portal/networks/wifi/leases
```

**Example response**:

```
HTTP/1.1 200 OK
Content-Type: application/json

[
  {
    "client_id": "01:5c:59:48:7d:b9:e6",
    "expires": "1511816276",
    "ip_addr": "192.168.128.64",
    "mac_addr": "5c:59:48:7d:b9:e6",
    "hostname": "paradrops-iPod"
```

```
    }
]
```

**get_network** (*request*, *chute*, *network*)
Get information about a network configured for the chute.

**Example request**:

```
GET /api/v1/chutes/captive-portal/networks/wifi
```

**Example response**:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "interface": "wlan0",
  "type": "wifi",
  "name": "wifi"
}
```

**get_networks** (*request*, *chute*)
Get list of networks configured for the chute.

**Example request**:

```
GET /api/v1/chutes/captive-portal/networks
```

**Example response**:

```
HTTP/1.1 200 OK
Content-Type: application/json

[
  {
    "interface": "wlan0",
    "type": "wifi",
    "name": "wifi"
  }
]
```

**get_ssid** (*request*, *chute*, *network*)
Get currently configured SSID for the chute network.

**Example request**:

```
GET /api/v1/chutes/captive-portal/networks/wifi/ssid
```

**Example response**:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "ssid": "Free WiFi",
  "bssid": "02:00:08:24:03:dd"
}
```

**get_station** (*request*, *chute*, *network*, *mac*)
Get detailed information about a connected station.

**Example request**:

```
GET /api/v1/chutes/captive-portal/networks/wifi/stations/5c:59:48:7d:b9:e6
```

**Example response**:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "rx_packets": "230",
  "tdls_peer": "no",
  "authenticated": "yes",
  "rx_bytes": "12511",
  "tx_bitrate": "1.0 MBit/s",
  "tx_retries": "0",
  "signal": "-45 [-49, -48] dBm",
  "authorized": "yes",
  "rx_bitrate": "65.0 MBit/s MCS 7",
  "mfp": "no",
  "tx_failed": "0",
  "inactive_time": "4688 ms",
  "mac_addr": "5c:59:48:7d:b9:e6",
  "tx_bytes": "34176",
  "wmm_wme": "yes",
  "preamble": "short",
  "tx_packets": "88",
  "signal_avg": "-44 [-48, -47] dBm"
}
```

**get_stations** (*request*, *chute*, *network*)

Get detailed information about connected wireless stations.

**Example request**:

```
GET /api/v1/chutes/captive-portal/networks/wifi/stations
```

**Example response**:

```
HTTP/1.1 200 OK
Content-Type: application/json

[
  {
    "rx_packets": "230",
    "tdls_peer": "no",
    "authenticated": "yes",
    "rx_bytes": "12511",
    "tx_bitrate": "1.0 MBit/s",
    "tx_retries": "0",
    "signal": "-45 [-49, -48] dBm",
    "authorized": "yes",
    "rx_bitrate": "65.0 MBit/s MCS 7",
    "mfp": "no",
    "tx_failed": "0",
    "inactive_time": "4688 ms",
    "mac_addr": "5c:59:48:7d:b9:e6",
    "tx_bytes": "34176",
    "wmm_wme": "yes",
    "preamble": "short",
    "tx_packets": "88",
```

```
            "signal_avg": "-44 [-48, -47] dBm"
        }
    ]
```

**hostapd_control**(*request*, *chute*, *network*)

**restart_chute**(*request*, *chute*)

**routes**

L{Klein} is an object which is responsible for maintaining the routing configuration of our application.

**@ivar _url_map: A C{werkzeug.routing.Map} object which will be used for** routing resolution.

@ivar _endpoints: A C{dict} mapping endpoint names to handler functions.

**set_chute_config**(*request*, *chute*)

Update the chute configuration and restart to apply changes.

**Example request**:

```
PUT /api/v1/chutes/captive-portal/config
Content-Type: application/json

{
  "net": {
    "wifi": {
      "dhcp": {
        "lease": "1h",
        "limit": 250,
        "start": 3
      },
      "intfName": "wlan0",
      "options": {
        "isolate": True
      },
      "ssid": "Better Free WiFi",
      "type": "wifi"
    }
  }
}
```

**Example response**:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "change_id": 1
}
```

**set_ssid**(*request*, *chute*, *network*)

Change the configured SSID for the chute network.

The change will not persist after a reboot. If a persistent change is desired, you should update the chute configuration instead.

**Example request**:

```
PUT /api/v1/chutes/captive-portal/networks/wifi/ssid
Content-Type: application/json

{
```

```
  "ssid": "Best Free WiFi"
}
```

**Example response**:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "message": "OK"
}
```

**start_chute**(*request*, *chute*)

**stop_chute**(*request*, *chute*)

**update_chute**(*request*, *chute*)

**class ChuteCacheEncoder**(*skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *encoding='utf-8'*, *default=None*)

Bases: json.encoder.JSONEncoder

JSON encoder for chute cache dictionary.

The chute cache can contain arbitrary objects, some of which may not be JSON-serializable. This encoder returns handles unserializable objects by returning the *repr* string.

**default**(*o*)

**class UpdateEncoder**(*skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *encoding='utf-8'*, *default=None*)

Bases: json.encoder.JSONEncoder

**default**(*o*)

**extract_tarred_chute**(*data*)

**tarfile_is_safe**(*tar*)

Check the names of files in the archive for safety.

Returns True if all paths are relative and safe or False if any of the paths are absolute (leading slash) or try to access parent directories (leading ..).

### paradrop.backend.config_api module

This module exposes device configuration.

Endpoints for these functions can be found under /api/v1/config.

**class ConfigApi**(*update_manager*, *update_fetcher*)

Bases: object

Configuration API.

This class handles HTTP API calls related to router configuration.

**factory_reset**(**args*, ***kwargs*)

Initiate the factory reset process.

**get_hostconfig**(*request*)

Get the device's current host configuration.

**Example request**:

```
GET /api/v1/config/hostconfig
```

**Example response**:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "firewall": {
    "defaults": {
      "forward": "ACCEPT",
      "input": "ACCEPT",
      "output": "ACCEPT"
    }
  },
  ...
}
```

For a complete example, please see the Host Configuration section.

**get_pdid**(*request*)
Get the device's current ParaDrop ID. This is the identifier assigned by the cloud controller.

**Example request**:

```
GET /api/v1/config/pdid
```

**Example response**:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  pdid: "5890e1e5ab7e317e6c6e049f"
}
```

**get_provision**(*request*)
Get the provision status of the device.

**pdconf**(*request*)
Get configuration sections from pdconf.

This returns a list of configuration sections and whether they were successfully applied. This is intended for debugging purposes.

**pdconf_reload**(*request*)
Trigger pdconf to reload UCI configuration files.

Trigger pdconf to reload UCI configuration files and return the status. This function is intended for low-level debugging of the paradrop pdconf module.

**provision**(*request*)
Provision the device with credentials from a cloud controller.

**routes**
L{Klein} is an object which is responsible for maintaining the routing configuration of our application.

**@ivar _url_map: A C{werkzeug.routing.Map} object which will be used for** routing resolution.

@ivar _endpoints: A C{dict} mapping endpoint names to handler functions.

**sshKeys**(*request*, *user*)
Manage list of authorized keys for SSH access.

**start_update**(*request*)

**update_hostconfig**(*request*)

Replace the device's host configuration.

**Example request**:

```
PUT /api/v1/config/hostconfig
Content-Type: application/json

{
  "firewall": {
    "defaults": {
      "forward": "ACCEPT",
      "input": "ACCEPT",
      "output": "ACCEPT"
    }
  },
  ...
}
```

**Example response**:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  change_id: 1
}
```

For a complete example, please see the Host Configuration section.

### paradrop.backend.cors module

Write the CROSS-ORIGIN RESOURCE SHARING headers required Reference: http://msoulier.wordpress.com/2010/06/05/cross-origin-requests-in-twisted/

**config_cors**(*request*)

### paradrop.backend.http_server module

The HTTP server to serve local portal and provide RESTful APIs

**class HttpServer**(*update_manager*, *update_fetcher*, *airshark_manager*, *portal_dir=None*)

Bases: object

**airshark_analyzer**(*request*, **args*, ***kwargs*)

**airshark_spectrum**(*request*, **args*, ***kwargs*)

**api_airshark**(*request*, **args*, ***kwargs*)

**api_auth**(*request*)

**api_changes**(*request*, **args*, ***kwargs*)

**api_chute**(*request*, **args*, ***kwargs*)

**api_configuration**(*request*, **args*, ***kwargs*)

**api_information**(*request*, **args*, ***kwargs*)

**api_password**(*request*, **args*, ***kwargs*)

**app**

L{Klein} is an object which is responsible for maintaining the routing configuration of our application.

**@ivar _url_map: A C{werkzeug.routing.Map} object which will be used for** routing resolution.

@ivar _endpoints: A C{dict} mapping endpoint names to handler functions.

**change_stream**(*request*, **args*, ***kwargs*)

**chute_logs**(*request*, **args*, ***kwargs*)

**home**(*request*, **args*, ***kwargs*)

**logs**(*request*, **args*, ***kwargs*)

**paradrop_logs**(*request*, **args*, ***kwargs*)

**snapd**(*request*, **args*, ***kwargs*)

**status**(*request*, **args*, ***kwargs*)

**annotate_routes**(*router*, *prefix*)

Annotate klein routes for compatibility with autoflask generator.

**setup_http_server**(*http_server*, *host*, *port*)

### paradrop.backend.information_api module

Provide information of the router, e.g. board version, CPU information, memory size, disk size.

Endpoints for these functions can be found under /api/v1/info.

**class InformationApi**

**get_environment**(*request*)

Get environment variables.

Returns a dictionary containing the environment variables passed to the Paradrop daemon. This is useful for development and debugging purposes (e.g. see how PATH is set on Paradrop when running in different contexts).

**Example request**:

```
GET /api/v1/info/environment
```

**Example response**:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "LANG": "C.UTF-8",
  "SNAP_REVISION": "x73",
  "SNAP_COMMON": "/var/snap/paradrop-daemon/common",
  "XDG_RUNTIME_DIR": "/run/user/0/snap.paradrop-daemon",
  "SNAP_USER_COMMON": "/root/snap/paradrop-daemon/common",
  "SNAP_LIBRARY_PATH": "/var/lib/snapd/lib/gl:/var/lib/snapd/void",
  "SNAP_NAME": "paradrop-daemon",
  "PWD": "/var/snap/paradrop-daemon/x73",
  "PATH": "/snap/paradrop-daemon/x73/usr/sbin:/snap/paradrop-daemon/x73/usr/bin:/snap/paradr
  "SNAP": "/snap/paradrop-daemon/x73",
```

```
  "SNAP_DATA": "/var/snap/paradrop-daemon/x73",
  "SNAP_VERSION": "0.9.2",
  "SNAP_ARCH": "amd64",
  "SNAP_USER_DATA": "/root/snap/paradrop-daemon/x73",
  "TEMPDIR": "/tmp",
  "HOME": "/root/snap/paradrop-daemon/x73",
  "SNAP_REEXEC": "",
  "LD_LIBRARY_PATH": "/var/lib/snapd/lib/gl:/var/lib/snapd/void:/snap/paradrop-daemon/x73/us
  "TMPDIR": "/tmp"
  ...
}
```

**get_features** (*request*)

Get features supported by the host.

This is a list of strings specifying features supported by the daemon.

**Explanation of feature strings**:

**hostapd-control** The daemon supports the hostapd control interface and provides a websocket channel for accessing it.

**Example request**:

```
GET /api/v1/info/features
```

**Example response**:

```
HTTP/1.1 200 OK
Content-Type: application/json

[
  "hostapd-control"
]
```

**get_telemetry** (*request*)

Get a telemetry report.

This contains information about resource utilization by chute and system totals. This endpoint returns the same data that we periodically send to the controller if telemetry is enabled.

**hardware_info** (*request*)

Get information about the hardware platform.

**Example request**:

```
GET /api/v1/info/hardware
```

**Example response**:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "wifi": [
    {
      "slot": "pci/0000:04:00.0",
      "vendorId": "0x168c",
      "macAddr": "04:f0:21:2f:b7:c1",
      "id": "pci-wifi-0",
      "deviceId": "0x003c"
    },
```

```
            {
              "slot": "pci/0000:06:00.0",
              "vendorId": "0x168c",
              "macAddr": "04:f0:21:0f:78:28",
              "id": "pci-wifi-1",
              "deviceId": "0x002a"
            }
          ],
          "memory": 2065195008,
          "vendor": "PC Engines",
          "board": "APU 1.0",
          "cpu": "x86_64"
        }
```

**routes**

L{Klein} is an object which is responsible for maintaining the routing configuration of our application.

**@ivar _url_map: A C{werkzeug.routing.Map} object which will be used for** routing resolution.

@ivar _endpoints: A C{dict} mapping endpoint names to handler functions.

**software_info**(*request*)

Get information about the operating system.

Returns a dictionary containing information the BIOS version, OS version, kernel version, Paradrop version, and system uptime.

**Example request**:

```
GET /api/v1/info/software
```

**Example response**:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "biosVersion": "SageBios_PCEngines_APU-45",
  "biosDate": "04/05/2014",
  "uptime": 15351,
  "kernelVersion": "Linux-4.4.0-101-generic",
  "pdVersion": "0.9.2",
  "biosVendor": "coreboot",
  "osVersion": "Ubuntu 4.4.0-101.124-generic 4.4.95"
}
```

### paradrop.backend.log_sockjs module

**class LogSockJSFactory**(*chutename*)

Bases: twisted.internet.protocol.Factory

**buildProtocol**(*addr*)

**class LogSockJSProtocol**(*factory*)

Bases: twisted.internet.protocol.Protocol

**check_log**()

**connectionLost**(*reason*)

**connectionMade**()

### paradrop.backend.password_api module

**class PasswordApi** (*password_manager*)
Bases: object

For now, we only support set/reset password for the default user: 'paradrop'

**change** (*request*)

**clear** (*request*)

**routes**
L{Klein} is an object which is responsible for maintaining the routing configuration of our application.

**@ivar _url_map: A C{werkzeug.routing.Map} object which will be used for** routing resolution.

@ivar _endpoints: A C{dict} mapping endpoint names to handler functions.

### paradrop.backend.password_manager module

**class PasswordManager**
Bases: object

**DEFAULT_PASSWORD = ''**

**DEFAULT_USER_NAME = 'paradrop'**

**add_user** (*user_name*, *password*)

**change_password** (*user_name*, *newPassword*)

**remove_user** (*user_name*)

**reset** ()

**verify_password** (*user_name*, *password*)

### paradrop.backend.snapd_resource module

**class SnapdResource**
Bases: twisted.web.resource.Resource

Expose the snapd API by forwarding requests.

https://github.com/snapcore/snapd/wiki/REST-API

**do_snapd_request** (*request*)
Forward the API request to snapd.

**isLeaf = True**

**render** (*request*)
Fulfill requests by forwarding them to snapd.

We use a synchronous implementation of HTTP over Unix sockets, so we do the request in a worker thread and have it call request.finish.

### paradrop.backend.status_sockjs module

**class** **StatusSockJSFactory**(*system_status*)
Bases: twisted.internet.protocol.Factory

**buildProtocol**(*addr*)

**class** **StatusSockJSProtocol**(*factory*)
Bases: twisted.internet.protocol.Protocol

**connectionLost**(*reason*)

**connectionMade**()

**dataReceived**(*data*)

### Module contents

## paradrop.base package

### Submodules

### paradrop.base.cxbr module

Wamp utility methods.

**class** **BaseClientFactory**(*factory*, **args*, ***kwargs*)
Bases: autobahn.twisted.websocket.WampWebSocketClientFactory, twisted.internet.protocol.ReconnectingClientFactory

**clientConnectionFailed**(*connector*, *reason*)

**clientConnectionLost**(*connector*, *reason*)

**initialDelay** = 1

**maxDelay** = 60

**class** **BaseSession**(*config=None*)
Bases: autobahn.twisted.wamp.ApplicationSession

Temporary base class for crossbar implementation

**call**(*procedure*, **args*, ***kwargs*)

**leave**()

**onJoin**(**args*, ***kwargs*)

**publish**(*topic*, **args*, ***kwargs*)

**register**(*endpoint*, *procedure=None*, *options=None*)

**classmethod** **start**(*klass*, *address*, *pdid*, *realm='paradrop'*, *start_reactor=False*, *debug=False*, *extra=None*, *reconnect=True*)
Creates a new instance of this session and attaches it to the router at the given address and realm.

**reconnect: The session will attempt to reconnect on connection failure** and continue trying indefinitely.

**stockCall**(*procedure*, **args*, ***kwargs*)

**stockPublish**(*topic*, **args*, ***kwargs*)

**stockRegister**(*endpoint*, *procedure=None*, *options=None*)

**stockSubscribe**(*handler*, *topic=None*, *options=None*)

**subscribe**(*handler*, *topic=None*, *options=None*)

**class BaseSessionFactory**(*config*, *deferred=None*)
Bases: autobahn.twisted.wamp.ApplicationSessionFactory

### paradrop.base.exceptions module

Exceptions and their subclasses

TODO: Distill these down and make a heirarchy.

**exception AuthenticationError**
Bases: *paradrop.base.exceptions.PdServerException*

**exception ChuteNotFound**
Bases: *paradrop.base.exceptions.ParadropException*

**exception ChuteNotRunning**
Bases: *paradrop.base.exceptions.ParadropException*

**exception InteralException**
Bases: exceptions.Exception

**exception InvalidCredentials**
Bases: *paradrop.base.exceptions.PdServerException*

**exception ModelNotFound**
Bases: *paradrop.base.exceptions.PdServerException*

**exception ParadropException**
Bases: exceptions.Exception

**exception PdServerException**
Bases: exceptions.Exception

**exception PdidError**
Bases: *paradrop.base.exceptions.PdServerException*

**exception PdidExclusionError**
Bases: *paradrop.base.exceptions.PdServerException*

### paradrop.base.nexus module

Stateful, singleton, paradrop daemon command center. See docstring for NexusBase class for information on settings.

**SETTINGS QUICK REFERENCE:** # assuming the following import from paradrop.base import nexus

nexus.core.info.version nexus.core.info.pdid

**class AttrWrapper**
Bases: object

Simple attr interceptor to make accessing settings simple.

Stores values in an internal dict called contents.

Does not allow modification once _lock() is called. Respect it.

Once you've filled it up with the appropriate initial values, set onChange to assign

**setOnChange**(*func*)

**class NexusBase**(*stealStdio=True*, *printToConsole=True*)
Bases: object

**Resolving these values to their final forms:** 1 - module imported, initial values assigned(as written below) 2 - class is instatiated, passed settings to replace values 3 - instance chooses appropriate values based on current state(production or local)

Each category has its own method for initialization here (see: resolveNetwork, resolvePaths)

**PDID = None**

**VERSION = 1**

**connect**(**args*, ***kwargs*)
Takes the given session class and attempts to connect to the crossbar fabric.

If an existing session is connected, it is cleanly closed.

**getKey**(*name*)
Returns the given key or None

**onInfoChange**(*key*, *value*)
Called when an internal setting is changed. Trigger a save automatically.

**onStart**()

**onStop**()

**provision**(*pdid*, *pdserver='https://paradrop.org'*, *wampRouter='ws://paradrop.org:9086/ws'*)

**provisioned**()
Checks if this[whatever] appears to be provisioned or not

**save**()
Ehh. Ideally this should happen asynchronously.

**saveKey**(*key*, *name*)
Save the key with the given name. Overwrites by default

**createDefaultInfo**(*path*)

**loadYaml**(*path*)
Return dict from YAML found at path

**resolveInfo**(*nexus*, *path*)
Given a path to the config file, load its contents and assign it to the config file as appropriate.

**validateInfo**(*contents*)
Error checking on the read YAML file. This is a temporary method.

**:**

**param contents:** the read - in yaml to check

**:**

**type contents:** dict.

**:**

**returns:** True if valid, else false

**writeYaml** (*contents*, *path*)
Overwrites content with YAML representation at given path

### paradrop.base.output module

Output mapping, capture, storange, and display.

Some of the methods and choice here may seem strange – they are meant to keep this file in

**class BaseOutput** (*logType*)
Bases: object

Base output type class.

This class and its subclasses are registered with an attribute on the global 'out' function and is responsible for formatting the given output stream and returning it as a "log structure" (which is a dict.)

**For example:** out.info("Text", anObject)

requires a custom object to figure out what to do with anObject where the default case will simply parse the string with an appropriate color.

Objects are required to output a dict that mininmally contains the keys message and type.

**formatOutput** (*logDict*)
Convert a logdict into a custom formatted, human readable version suitable for printing to console.

**class ExceptionOutput** (*logType*)
Bases: *paradrop.base.output.BaseOutput*

Handle vanilla exceptions passed directly to us using out.exception

**class Level**
Bases: enum.Enum

**ERR = <Level.ERR: 6>**

**FATAL = <Level.FATAL: 8>**

**HEADER = <Level.HEADER: 1>**

**INFO = <Level.INFO: 3>**

**PERF = <Level.PERF: 4>**

**SECURITY = <Level.SECURITY: 7>**

**USAGE = <Level.USAGE: 9>**

**VERBOSE = <Level.VERBOSE: 2>**

**WARN = <Level.WARN: 5>**

**class Output** (***kwargs*)

Class that allows stdout/stderr trickery. By default the paradrop object will contain an @out variable (defined below) and it will contain 2 members of "err" and "fatal".

Each attribute of this class should be a function which points to a class that inherits IOutput(). We call these functions "output streams".

The way this Output class is setup is that you pass it a series of kwargs like (stuff=OutputClass()). Then at any point in your program you can call "paradrop.out.stuff('This is a string

')".

This way we can easily support different levels of verbosity without the need to use some kind of bitmask or anything else. On-the-fly output creation is no longer supported due to the metadata and special processing added. It is still possible, but not implemented.

This is done by the __getattr__ function below, basically in __init__ we set any attributes you pass as args, and anything else not defined gets sent to __getattr__ so that it doesn't error out.

**endLogging**()
Ask the printing thread to flush and end, then return.

**getLogsSince**(*target*, *purge=False*)
Reads all logs and returns their contents. The current log file is not touched. Removes old log files if 'purge' is set (though this is a topic for debate...)

The server will be most interested in this call, but it needs to register for new logs first, else there's a good chance to see duplicates.

NOTE: don't open all log files, check to open only the ones that might be relevant. This is certainly a bug and can cause memory issues.

**Parameters**

- **target** (*float.*) – seconds since the GMT epoch. Method returns logs that have timestamps later than this.
- **purge** (*bool.*) – deletes the old log files (except today's) if set

**Returns** a list of dictionaries containing log information. Not ordered.

**handlePrint**(*logDict*)
All printing objects return their messages. These messages are routed to this method for handling.

Send the messages to the printer. Optionally display the messages. Decorate the print messages with metadata.

**Parameters** **logDict** – a dictionary representing this log item. Must contain keys

message and type. :type logDict: dict.

**logToConsole**(*newStatus*)

**messageToString**(*message*)
Converts message dicts to a format suitable for printing based on the conversion rules laid out in in that class's implementation.

**Parameters** **message** (*dict.*) – the dict to convert to string

**Returns** str

**startLogging**(*filePath=None*, *stealStdio=False*, *printToConsole=True*)
Begin logging. The output class is ready to go out of the box, but in order to prevent mere imports from stealing stdio or console logging to vanish these must be manually turned on.

**Parameters**

- **filePath** (*str*) – if provided, begin logging to the given directory. If not provided, do not write out logs.
- **stealStdio** (*bool.*) – choose to intercept stdio (including vanilla print statements) or allow it to passthrough
- **printToConsole** (*bool.*) – output the results of all logging to the console. This is primarily a performance consideration when running in production

**stealStdio**(*newStatus*)

**class** `OutputRedirect`(*output*, *contentAppearedCallback*, *logType*)
Bases: `object`

Intercepts passed output object (either stdout and stderr), calling the provided callback method when input appears.

Retains the original mappings so writing can still happen. Performs no formatting.

`flush`()

`trueWrite`(*contents*)
Someone really does want to output

`write`(*contents*)
Intercept output to the assigned target and callback with it. The true output is returned with the callback so the delegate can differentiate between captured outputs in the case when two redirecters are active.

**class** `PrintLogThread`(*path*, *queue*, *name*)
Bases: `threading.Thread`

All file printing access from one thread.

Receives information when its placed on the passed queue. Called from one location: Output.handlePrint.

Does not close the file: this happens in Output.endLogging. This simplifies the operation of this class, since it only has to concern itself with the queue.

The path must exist before DailyLog runs for the first time.

`run`()

**class** `TwistedException`(*logType*)
Bases: *paradrop.base.output.BaseOutput*

**class** `TwistedOutput`(*logType*)
Bases: *paradrop.base.output.BaseOutput*

`blacklist` = **['Starting factory', 'Stopping factory', 'Log opened']**

`parseLogPrefix`(*tb*)
Takes a traceback returned by 'extract_tb' and returns the package, module, and line number

`silentLogPrefix`(*stepsUp*)
logPrefix v2– gets caller information silently (without caller intervention) The single parameter reflects how far up the stack to go to find the caller and depends how deep the direct caller to this method is wrt to the target caller

NOTE: Some calls cannot be silently prefixed (getting into the twisted code is a great example)

**Parameters** `stepsUp` – the number of steps to move up the stack for the caller

### paradrop.base.pdutils module

lib.utils.output. Helper for formatting output from Paradrop.

**class** `Timer`(*key=''*, *verbose=True*)
Bases: `object`

A timer object for simple benchmarking.

**Usage:**

**with Timer(key='Name of this test') as t:** do.someCode(thatTakes=aWhile)

Once the code finishes executing the time is output.

**check**(*pkt*, *pktType*, *keyMatches=None*, ***valMatches*)
This function takes an object that was expected to come from a packet (after it has been JSONized) and compares it against the arg requirements so you don't have to have 10 if() statements to look for keys in a dict, etc..

**Args:** @pkt : object to look at @pktType : object type expected (dict, list, etc..) @keyMatches : a list of minimum keys found in parent level of dict, expected to be an array @valMatches : a dict of key:value pairs expected to be found in the parent level of dict

the value can be data (like 5) OR a type (like this value must be a @list@).

**Returns:** None if everything matches, otherwise it returns a string as to why it failed.

**convertUnicode**(*elem*)
Converts all unicode strings back into UTF-8 (str) so everything works. Call this function like:

json.loads(s, object_hook=convertUnicode)

**class dict2obj**(*aDict=None*, ***kwargs*)
Bases: object

**explode**(*pkt*, **args*)
This function takes a dict object and explodes it into the tuple requested.

It returns None for any value it doesn't find.

The only error it throws is if args is not defined.

**Example:** pkt = {'a':0, 'b':1} 0, 1, None = pdcomm.explode(pkt, 'a', 'b', 'c')

**json2str**(*j*, *safe=' '*)
Properly converts and encodes all data related to the JSON object into a string format that can be transmitted through a network and stored properly in a database. Arguments:

@j : json to be converted @safe : optional, string of chars to pass to urlEncodeMe that are declared safe (don't encode)

**jsonPretty**(*j*)
Returns a string of a JSON object in 'pretty print' format fully indented, and sorted.

**stimestr**(*x=None*)

**str2json**(*s*)

**timedur**(*x*)
Print consistent string format of seconds passed. Example: 300 = '5 mins' Example: 86400 = '1 day' Example: 86705 = '1 day, 5 mins, 5 sec'

**timeflt**()

**timeint**()

**timestr**(*x=None*)

**urlDecodeMe**(*elem*)
Converts any values that would cause JSON parsing to fail into URL percent encoding equivalents. This function can be used for any valid JSON type including str, dict, list. Returns:

Same element properly decoded.

**urlEncodeMe**(*elem*, *safe=' '*)
Converts any values that would cause JSON parsing to fail into URL percent encoding equivalents. This function can be used for any valid JSON type including str, dict, list. Returns:

Same element properly encoded.

### paradrop.base.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 KYE:VALUE array

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

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

**loadSettings**(*mode='local'*, *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

**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.

**updatePaths**(*configHomeDir*, *runtimeHomeDir='/var/run/paradrop'*)

### Module contents

## paradrop.confd package

### Submodules

### paradrop.confd.base module

**class ConfigObject**(*name=None*)
Bases: object

**PRIO_CONFIG_IFACE = 30**

**PRIO_CONFIG_QDISC = 45**

**PRIO_CREATE_IFACE = 20**

**PRIO_CREATE_QDISC = 40**

**PRIO_CREATE_VLAN = 25**

**PRIO_IPTABLES_RULE = 37**

**PRIO_IPTABLES_TOP = 35**

**PRIO_IPTABLES_ZONE = 36**

**PRIO_START_DAEMON = 60**

**apply**(*allConfigs*)
Return a list of commands to apply this configuration.

Most subclasses will need to implement this function.

Returns a list of (priority, Command) tuples.

**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

**copy**()
Make a copy of the config object.

The copy will receive the same name and option values.

**dump**()
Return full configuration section as a string.

**findByType**(*allConfigs*, *module*, *typename*, *where={}*)
Look up sections by type (generator).

where: filter the returned results by checking option values.

**classmethod** **getModule**()
Get the module name (e.g. "dhcp", "wireless") for a ConfigObject class.

**getName**()
Return section name.

Subclasses that do not have names (anonymous sections) should override this to return some other unique identifier such as an interface name.

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

**lookup**(*allConfigs*, *sectionModule*, *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.

**maskable = True**

**nextId = 0**

**options = []**

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

**static** **prioritizeConfigs**(*configs*, *reverse=False*)
Assign priorities to config objects based on the dependency graph.

Priority zero is assigned to all configs with no dependencies.

priority(config1) > priority(config2) means config1 should be applied later than config2, and config1 should be reverted earlier than config2. For configs with the same priority value, it is presumed that order does not matter.

If reverse is True, the priorities are made negative so that traversing in increasing order gives the proper order for reverting.

Returns a list of tuples (priority, config). This format is suitable for heapq.

**removeFromParents**()
Remove this section from being tracked by its parents.

Call this before discarding a configuration section so that later on, if the parent is updated, it doesn't try to update non-existent children.

**revert**(*allConfigs*)
Return a list of commands to revert this configuration.

Most subclasses will need to implement this function.

Returns a list of (priority, Command) tuples.

**setup**()
Finish object initialization.

This is called after the config object is initialized will all of its options values filled in. Override to do some preparation work before we start generating commands.

**typename = None**

**updateApply**(*new*, *allConfigs*)
Return a list of commands to update to new configuration.

Implementing this is optional for subclasses. The default behavior is to call apply.

Returns a list of (priority, Command) tuples.

**updateRevert**(*new*, *allConfigs*)
Return a list of commands to (partially) revert the configuration.

The implementation can be selective about what it reverts (e.g. do not delete an interface if we are only updating its IP address). The default behavior is to call revert.

Returns a list of (priority, Command) tuples.

**class ConfigOption**(*name*, *type=<type 'str'>*, *required=False*, *default=None*)
Bases: object

**default**

**name**

**required**

**type**

**interpretBoolean**(*s*)
Interpret string as a boolean value.

"0" and "False" are interpreted as False. All other strings result in True. Technically, only "0" and "1" values should be seen in UCI files. However, because of some string conversions in Python, we may encounter "False".

### paradrop.confd.client module

**reload**(*path*)
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 it will return None.

**reloadAll**()
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 it will return None.

**systemStatus**()
Return system status string from pdconf.

**waitSystemUp**()
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 it will return None.

### paradrop.confd.command module

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

**execute**()

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

**class CommandList**
Bases: list

**append**(*priority*, *command*)

**commands**()
Iterate over commands in order by priority.

Commands are first sorted by assigned priority. Within each priority level, the order in which they were added is maintained.

**class KillCommand**(*pid*, *parent=None*)
Bases: *paradrop.confd.command.Command*

Special command object for killing a process

**execute**()

**getPid**()

**kill**(*pid*, *kill_signal=4*, *timeout=8*)
Kill a child process and wait with timeout.

1.Send a SIGTERM signal to the process.

2.Wait up to *kill_signal* seconds for the process to exit.

3.If process is still running, send a SIGKILL signal.

4. Wait up to *timeout* seconds (cumulative with *kill_signal*) for the process to exit.

Returns True if the process exited before *timeout* seconds elapsed.

### paradrop.confd.dhcp module

**class ConfigDhcp**(*name=None*)
Bases: *paradrop.confd.base.ConfigObject*

**options = [ConfigOption(name='interface', type=<type 'str'>, required=True, default=None), ConfigOption(name='lea**

**typename = 'dhcp'**

class **ConfigDnsmasq**(*name=None*)
Bases: *paradrop.confd.base.ConfigObject*

**apply**(*allConfigs*)

**options** = [ConfigOption(name='authoritative', type=<type 'bool'>, required=False, default=True), ConfigOption(name

**revert**(*allConfigs*)

**typename** = 'dnsmasq'

class **ConfigDomain**(*name=None*)
Bases: *paradrop.confd.base.ConfigObject*

**getName**()

**options** = [ConfigOption(name='name', type=<type 'str'>, required=False, default=None), ConfigOption(name='ip', ty

**typename** = 'domain'

### paradrop.confd.firewall module

class **ConfigDefaults**(*name=None*)
Bases: *paradrop.confd.base.ConfigObject*

**apply**(*allConfigs*)

**getName**()

**get_iptables**()
Get the list of iptables commands to use (iptables / ip6tables).

**options** = [ConfigOption(name='input', type=<type 'str'>, required=False, default='ACCEPT'), ConfigOption(name='

**revert**(*allConfigs*)

**typename** = 'defaults'

**updateApply**(*new*, *allConfigs*)

**updateRevert**(*new*, *allConfigs*)

class **ConfigForwarding**(*name=None*)
Bases: *paradrop.confd.base.ConfigObject*

**apply**(*allConfigs*)

**options** = [ConfigOption(name='src', type=<type 'str'>, required=True, default=None), ConfigOption(name='dest', typ

**revert**(*allConfigs*)

**typename** = 'forwarding'

class **ConfigRedirect**(*name=None*)
Bases: *paradrop.confd.base.ConfigObject*

**ANY_PROTO** = set(['none', None, 'any'])

**apply**(*allConfigs*)

**options** = [ConfigOption(name='src', type=<type 'str'>, required=False, default=None), ConfigOption(name='src_ip', t

**revert**(*allConfigs*)

**typename** = 'redirect'

**class ConfigRule**(*name=None*)
Bases: *paradrop.confd.base.ConfigObject*

**apply**(*allConfigs*)

**get_iptables**()
Get the list of iptables commands to use (iptables / ip6tables).

**options = [ConfigOption(name='name', type=<type 'str'>, required=False, default=None), ConfigOption(name='src', ty**

**revert**(*allConfigs*)

**typename = 'rule'**

**class ConfigZone**(*name=None*)
Bases: *paradrop.confd.base.ConfigObject*

**apply**(*allConfigs*)

**get_iptables**()
Get the list of iptables commands to use (iptables / ip6tables).

**options = [ConfigOption(name='name', type=<type 'str'>, required=True, default=None), ConfigOption(name='networ**

**revert**(*allConfigs*)

**setup**()

**typename = 'zone'**

### paradrop.confd.main module

This module listens for 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.

**listen**(*configManager*)

**run_thread**(*execute=True*)
Start pdconfd service as a thread.

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

### paradrop.confd.manager module

**class ConfigManager**(*writeDir*, *execCommands=True*)
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*)
Execute commands.

Takes a CommandList object.

**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.

### paradrop.confd.network module

**class ConfigInterface**(*name=None*)
Bases: *paradrop.confd.base.ConfigObject*

**DEV_PLUS_VID = <_sre.SRE_Pattern object>**

**addToBridge**(*ifname*)
Generate commands to add ifname to bridge.

**apply**(*allConfigs*)

**maskable = False**

**options = [ConfigOption(name='proto', type=<type 'str'>, required=True, default=None), ConfigOption(name='ifname**

**removeFromBridge**(*ifname*)
Generate commands to add ifname to bridge.

**revert**(*allConfigs*)

**setup**()

**typename = 'interface'**

**updateApply**(*new*, *allConfigs*)

**updateRevert**(*new*, *allConfigs*)

### paradrop.confd.qos module

**class ConfigClass**(*name=None*)
Bases: *paradrop.confd.base.ConfigObject*

**options = [ConfigOption(name='packetsize', type=<type 'int'>, required=False, default=None), ConfigOption(name='p**

**typename = 'class'**

**class ConfigClassgroup**(*name=None*)
Bases: *paradrop.confd.base.ConfigObject*

**get_class_id**(*class_name*)
Get ID for a traffic class in this group.

Returns None if the class is not a member of the group.

**options = [ConfigOption(name='classes', type=<type 'str'>, required=True, default=None), ConfigOption(name='defau**

**setup**()

**typename = 'classgroup'**

**class ConfigClassify**(*name=None*)
Bases: *paradrop.confd.base.ConfigObject*

**apply**(*allConfigs*)

**make_iptables_cmd**(*action*, *ifname*, *class_id*)

**options = [ConfigOption(name='target', type=<type 'str'>, required=True, default=None), ConfigOption(name='proto'**

**revert**(*allConfigs*)

**typename = 'classify'**

**class ConfigInterface**(*name=None*)
Bases: *paradrop.confd.base.ConfigObject*

**apply**(*allConfigs*)

**options = [ConfigOption(name='enabled', type=<type 'bool'>, required=True, default=None), ConfigOption(name='cla**

**revert**(*allConfigs*)

**typename = 'interface'**

**compute_hfsc_params**(*classes*, *capacity*)

### paradrop.confd.wireless module

**class ConfGenerator**
Bases: object

**writeHeader**(*output*)

**writeOptions**(*options*, *output*, *title=None*)

**class ConfigWifiDevice**(*name=None*)
Bases: *paradrop.confd.base.ConfigObject*

**apply**(*allConfigs*)

**detectPrimaryInterface**()
Find the primary network interface associated with this Wi-Fi device.

By primary we mean the first interface (e.g. wlan0 or wlan1) that exists at system startup before any *interface add* commands. We will use the primary interface first, and create additional virtual interfaces after that.

That seems overly complicated, but it is required in cases where the Wi-Fi device does not support virtual interfaces.

Returns interface name or None.

**nextInterfaceName**()
Get the next available interface name.

**options = [ConfigOption(name='type', type=<type 'str'>, required=True, default=None), ConfigOption(name='phy', ty**

**releaseInterfaceName**(*ifname*)
Mark an interface name as no longer used.

**revert**(*allConfigs*)

**setup**()

**typename = 'wifi-device'**

**class ConfigWifiIface**(*name=None*)
Bases: *paradrop.confd.base.ConfigObject*

**apply**(*allConfigs*)

**getIfname**(*device*, *interface*)
Returns the name to be used by this WiFi interface, e.g. as seen by ifconfig.

This comes from the "ifname" option if it is set. Otherwise, we use the interface name of the associated network.

**getName**()
Return a unique and consistent identifier for the section.

If ifname is set, then that is a good choice for the name because interface names need to be unique on the system.

If ifname is not set, then we use the combined string device:network. The assumption is that no one will put multiple APs on the same device and same network, or if they do, (e.g. multiple APs on the br-lan bridge), then they will configure the ifname to be unique.

**getRandomMAC**()
Generate a random MAC address.

Returns a string "02:xx:xx:xx:xx:xx". The first byte is 02, which indicates a locally administered address.

**makeHostapdConf**(*wifiDevice*, *interface*)

**makeWpaSupplicantConf**(*wifiDevice*, *interface*)

**options = [ConfigOption(name='device', type=<type 'str'>, required=True, default=None), ConfigOption(name='mode'**

**revert**(*allConfigs*)

**typename = 'wifi-iface'**

**updateApply**(*new*, *allConfigs*)

**updateRevert**(*new*, *allConfigs*)

**class HostapdConfGenerator**(*wifiIface*, *wifiDevice*, *interface*)
Bases: *paradrop.confd.wireless.ConfGenerator*

**generate**(*path*)

**get11acOptions**()

**get11nOptions**()

**get11rOptions**()
Get options related to 802.11r (fast BSS transition).

**getMainOptions**()

**getRadiusOptions**()

**getSecurityOptions**()

**readMode**(*device*)
Determine HT/VHT mode if applicable.

**writeHeader**(*output*)

**class WpaSupplicantConfGenerator**(*wifiIface*, *wifiDevice*, *interface*)
Bases: *paradrop.confd.wireless.ConfGenerator*

**generate**(*path*)

**getMainOptions**()

**writeHeader**(*output*)

**getPhyFromMAC**(*mac*)

**getPhyMACAddress**(*phy*)

**get_cipher_list**(*encryption_mode*)
Get list of ciphers from encryption mode.

Example: get_cipher_list("psk2+tkip+aes") -> ["TKIP", "CCMP"]

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

### Module contents

## paradrop.core package

### Subpackages

#### paradrop.core.agent package

#### Submodules

**paradrop.core.agent.http module**

**class CurlRequestDriver**
Bases: *paradrop.core.agent.http.HTTPRequestDriver*

**code_pattern = <_sre.SRE_Pattern object>**

**curl = <MagicMock name='mock.Curl()' id='139871631477392'>**

**header_pattern = <_sre.SRE_Pattern object>**

**lock = <twisted.internet.defer.DeferredLock object>**

**receive**(*ignore*)
Receive response from curl and convert it to a response object.

**receiveHeaders**(*header_line*)

**request**(*method*, *url*, *body=None*)

**class HTTPRequestDriver**
Bases: object

**request**(*method*, *url*, *body*)

**setHeader**(*key*, *value*)

**class HTTPResponse**(*data=None*)
Bases: object

**class JSONReceiver**(*response*, *finished*)
Bases: twisted.internet.protocol.Protocol

JSON Receiver

A JSONReceiver object can be used with the twisted HTTP client to receive data from a request and provide it to a callback function when complete.

Example (response came from an HTTP request): finished = Deferred() response.deliverBody(JSONReceiver(finished)) finished.addCallback(func_that_takes_result)

Some error conditions will result in the callback firing with a result of None. The receiver needs to check for this. This seems to occur on 403 errors where the server does not return any data, but twisted just passes us a ResponseDone object the same type as a normal result.

**connectionLost**(*reason*)
internal: handles connection close events.

**dataReceived**(*data*)
internal: handles incoming data.

**class PDServerRequest**(*path*, *driver=<class 'paradrop.core.agent.http.CurlRequestDriver'>*, *setAuthHeader=True*)
Bases: object

Make an HTTP request to pdserver.

The API is assumed to use application/json for sending and receiving data. Authentication is automatically handled here if the router is provisioned.

We handle missing, invalid, or expired tokens by making the request and detecting a 401 (Unauthorized) response. We request a new token and retry the failed request. We do this at most once and return failure if the second attempt returns anything other than 200 (OK).

PDServerRequest objects are not reusable; create a new one for each request.

URL String Substitutions: router_id -> router id

Example: /routers/{router_id}/states -> /routers/halo06/states

**get**(***query*)

**classmethod getServerInfo**(*c*)
Return the information needed to send API messages to the server.

This can be used by an external program (e.g. pdinstall).

**patch**(**ops*)
Expects a list of operations in jsonpatch format (http://jsonpatch.com/).

An example operation would be: {'op': 'replace', 'path': '/completed', 'value': True}

**post**(***data*)

**put**(***data*)

**receiveResponse**(*response*)
Intercept the response object, and if it's a 401 authenticate and retry.

**request**()

**classmethod resetToken**(*c*)
Reset the auth token, to be called if the router's identity has changed.

**token = None**

**class PDServerResponse**(*response*, *data=None*)
Bases: object

A PDServerResponse object contains the results of a request to pdserver.

This wraps twisted.web.client.Response (cannot be subclassed) and exposes the same variables in addition to a 'data' variables. The 'data' variable, if not None, is the parsed object from the response body.

**class TwistedRequestDriver**
Bases: *paradrop.core.agent.http.HTTPRequestDriver*

**pool = <twisted.web.client.HTTPConnectionPool object>**

**receive**(*response*)
Receive response from twisted web client and convert it to a PDServerResponse object.

**request**(*method*, *url*, *body=None*)

**sem = <twisted.internet.defer.DeferredSemaphore object>**

**urlEncodeParams**(*data*)
Return data URL-encoded.

This function specifically handles None and boolean values to convert them to JSON-friendly strings (e.g. None -> 'null').

**paradrop.core.agent.reporting module**

class **ReportSender**(*model='states'*, *max_retries=None*)
Bases: object

**increaseDelay**()

**send**(*report*)

class **StateReport**
Bases: object

**toJSON**()

class **StateReportBuilder**
Bases: object

**prepare**()

class **TelemetryReportBuilder**
Bases: object

**prepare**()

**sendStateReport**()

**sendTelemetryReport**()

**paradrop.core.agent.wamp_session module** The WAMP session of the paradrop daemon

class **WampSession**(**args*, ***kwargs*)
Bases: *paradrop.base.cxbr.BaseSession*

**onChallenge**(*challenge*)

**onConnect**()

**onDisconnect**()

**onJoin**(**args*, ***kwargs*)

**onLeave**(*details*)

classmethod **set_update_fetcher**(*update_fetcher*)

**update**(*pdid*, *data*)

**update_fetcher** = None

**updatesPending**(**args*, ***kwargs*)

**Module contents**

**paradrop.core.chute package**

**Submodules**

**paradrop.core.chute.chute module**

**class Chute**(*descriptor*, *strip=None*)

Bases: object

Wrapper class for Chute objects.

**CONFIG_FIELDS = set(['environment', 'web', 'net', 'host_config'])**

**STATE_DISABLED = 'disabled'**

**STATE_FROZEN = 'frozen'**

**STATE_INVALID = 'invalid'**

**STATE_RUNNING = 'running'**

**STATE_STOPPED = 'stopped'**

**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.

**getCache**(*key*)

Get the val out of the _cache dict object, or None if it doesn't exist.

**getCacheContents**()

Return the cache dictionary.

**getConfiguration**()

Get the chute's configuration object.

**getHostConfig**()

Get the chute's host_config options for Docker.

Returns an empty dictionary if there is no host_config setting.

**getWebPort**()

Get the port configured for the chute's web server.

Returns port (int) or None if no port is configured.

**isRunning**()

**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.core.chute.chute_storage module**

**class ChuteStorage**(*filename=None*, *save_timer=0*)

Bases: *paradrop.lib.utils.pd_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 names 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.core.chute.restart module** 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

**Module contents**

### paradrop.core.config package

**Submodules**

**paradrop.core.config.airshark module**
**class AirsharkInterfaceManager**
Bases: object

**add_observer**(*observer*)

**interface_available**()

**remove_observer**(*observer*)

**reset_interface**()

**set_interface**(*interface*)

**configure**(*update*)

#### paradrop.core.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*)

#### paradrop.core.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.

**class SysReader**(*phy*)

Bases: object

**PCI_BUS_ID = <_sre.SRE_Pattern object>**

**USB_BUS_ID = <_sre.SRE_Pattern object>**

**getDeviceId**(*default='????'*)

Return the device ID for the device.

This is a four-digit hexadecimal number. For example, our Qualcomm 802.11n chips have device ID 002a.

**getSlotName**(*default='????'*)

Return the PCI/USB slot name for the device.

Example: "pci/0000:04:00.0" or "usb/1-1:1.0"

**getVendorId**(*default='????'*)

Return the vendor ID for the device.

This is a four-digit hexadecimal number. For example, our Qualcomm 802.11n chips have vendor ID 168c.

**read_uevent**()

Read the device uevent file and return the contents as a dictionary.

**class UCIBuilder**

Bases: object

UCIBuilder helps aggregate UCI configuration sections for writing to files.

**FILES = ['dhcp', 'network', 'firewall', 'wireless', 'qos']**

**add**(*file_*, *type_*, *options*, *name=None*)

Add a new configuration section.

**getSections**(*file_*)

Get sections associated with a single file.

Returns: list of tuples, [(config, options)]

**write**()
Write all of the configuration sections to files.

**checkSystemDevices**(*update*)
Check whether expected devices are present.

This may reboot the machine if devices are missing and the host config is set to do that.

**detectSystemDevices**()
Detect devices on the system.

The result is three lists stored in a dictionary. The three lists are indexed by 'wan', 'wifi', and 'lan'. Other devices may be supported by adding additional lists.

Within each list, a device is represented by a dictionary. For all devices, the 'name' and 'mac' fields are defined. For WiFi devices, the 'phy' is defined in addition. Later, we may fill in more device information (e.g. what channels a WiFi card supports).

**flushWirelessInterfaces**(*phy*)
Remove all virtual interfaces associated with a wireless device.

This should be used before giving a chute exclusive access to a device (e.g. monitor mode), so that it does not inherit unexpected interfaces.

**getMACAddress**(*ifname*)

**getPhyMACAddress**(*phy*)

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

Store device information in cache key "networkDevices" as well as "networkDevicesByName".

**getWirelessPhyName**(*ifname*)

**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.

**listSystemDevices**()
Detect devices on the system.

The result is a single list of dictionaries, each containing information about a network device.

**listWiFiDevices**()

**readHostconfigVlan**(*vlanInterfaces*, *builder*)

**readHostconfigWifi**(*wifi*, *networkDevices*, *builder*)

**readHostconfigWifiInterfaces**(*wifiInterfaces*, *networkDevices*, *builder*)

**readSysFile**(*path*)

**resetWirelessDevice**(*phy*, *primary_interface*)
Reset a wireless device's interfaces to clean state.

This will rename, delete, or add an interface as necessary to make sure only the primary interface exists, e.g. "wlan0" for a wireless device, e.g. phy0.

**resolveWirelessDevRef**(*name*, *networkDevices*)
Resolve a WiFi device reference (wlan0, phy0, 00:11:22:33:44:55, etc.) to the name of the device section as used by pdconf (wifiXXXXXXXXXXXX).

Unambiguous naming is preferred going forward (either wifiXX or the MAC address), but to maintain backward compatibility, we attempt to resolve either wlanX or phyX to the MAC address of the device that currently uses that name.

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

**setSystemDevices**(*update*)
Initialize system configuration files.

This section should only be run for host configuration updates.

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

### paradrop.core.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.core.config.dockerconfig module

**dockerconfig module:** This module contains all of the knowledge of how to take internal pdfcd representation of configurations of chutes and translate them into specifically what docker needs to function properly, whether that be in the form of dockerfiles or the HostConfig JSON object known at init time of the chute.

**abortCreateVolumeDirs**(*update*)

**createVolumeDirs**(*update*)

**generateToken**(*bits=128*)

**getVirtPreamble**(*update*)

### paradrop.core.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.core.config.haproxy module** This module is responsible for configuration haproxy.

**generateConfigSections**()

**reconfigureProxy**(*update*)

**writeConfigFile**(*output*)

**paradrop.core.config.hostconfig module** The host configuration controls system settings of the host OS.

This module operates as follows:

1. The first time, we try to detect all devices and auto-generate a reasonable configuration, which we store to a persistent file.

2. (TODO) We present the configuration to the owner sometime around provisioning or first chute creation and allow him to change settings.

3. (TODO) We have some kind of update operation that can manipulate settings.

**generateHostConfig**(*devices*)
Scan for devices on the machine and generate a working configuration.

**getHostConfig**(*update*)
Load host configuration.

Read device information from networkDevices. Store host configuration in hostConfig.

**load**(*path=None*)
Load host configuration.

Tries to load host configuration from persistent file. If that does not work, it will try to automatically generate a working configuration.

Returns a host config object on success or None on failure.

**prepareHostConfig**(*devices=None*, *hostConfigPath=None*, *write=True*)
Load an existing host configuration or generate one.

Tries to load host configuration from persistent file. If that does not work, it will try to automatically generate a working configuration.

write: if True and host config was automatically generated, then write the new host config to a file.

**revertHostConfig**(*update*)
Restore host configuration from before update.

Uses oldHostConfig cache entry.

**save**(*config*, *path=None*)
Save host configuration.

May raise exception if unable to write the configuration file.

**setAutoUpdate**(*enable*)

**setHostConfig**(*update*)
Write host configuration to persistent storage.

Read host configuration from hostConfig.

**paradrop.core.config.network module**

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

**chooseExternalIntf**(*update*, *iface*)

**chooseSubnet**(*update*, *cfg*, *iface*)

**fulfillDeviceRequest**(*update*, *cfg*, *devices*)
Find a physical device that matches the requested device type.

Raises an exception if one cannot be found.

**getExtraOptions**(*cfg*)
Get dictionary of extra wifi-iface options that we are not interpreting but just passing on to pdconf.

**getInterfaceAddress**(*update*, *name*, *cfg*, *iface*)
Dynamically select IP address for the chute interface.

This function will use a subnet from the chute subnet pool and assign IP addresses to the external (in host) and internal (in chute) interfaces.

The addresses are stored in the iface object.

**getInterfaceDict**(*chute*)
Return interfaces from a chute as a dict with interface names as the keys. Returns an empty dict if chute is None or it had no interfaces.

**getL3BridgeConfig**(*update*)
Creates configuration sections for layer 3 bridging.

**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...

Store configuration in networkInterfaces cache entry.

**getNetworkConfigLan**(*update*, *name*, *cfg*, *iface*)

**getNetworkConfigVlan**(*update*, *name*, *cfg*, *iface*)

**getNetworkConfigWifi**(*update*, *name*, *cfg*, *iface*)

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

**getWifiKeySettings**(*cfg*, *iface*)
Read encryption settings from cfg and transfer them to iface.

**get_current_phy_conf**(*update*, *device_id*)
Lookup current configuration for a network device.

This includes information such as the Wi-Fi channel.

Returns a dictionary, which may be empty if no configuration was found.

**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.

**satisfies_requirements**(*obj*, *requirements*)
Checks that an object satifies given requirements.

Every key-value pair in the requirements object must be present in the target object for it to be considered satisfied.

Returns True/False.

**setL3BridgeConfig**(*update*)
Apply configuration for layer 3 bridging.

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

#### paradrop.core.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.core.config.power module

**reboot**(*update*)
**shutdown**(*update*)

#### paradrop.core.config.reservations module

Module for checking resource reservations by chutes.

One idea motivating this design is to reduce the amount of state in memory for resource reservations. We have the chute list, which contains information about what devices the chute is using. If we also maintain a separate list of devices used by chutes, we need to keep them synchronized. This becomes messy when a chute fails to install or uninstall correctly. The getDeviceReservations function iterates over the chute list and returns an up-to-date view of device usage. This can be called as needed.

**class DeviceReservations**
Bases: object

**add**(*chute*, *dtype*, *mode=None*)

**count**(*dtype=None*, *mode=None*)
Return the number of reservations matching the given criteria.

None is used as a wildcard, so if no arguments are passed, the count returned is the total number of reservations.

**class InterfaceReservationSet**
Bases: object

**add**(*interface*)

**class SubnetReservationSet**
Bases: object

**add**(*subnet*)

**getDeviceReservations**(*exclude=None*)
Produce a dictionary mapping device names to DeviceReservations objects that describe the current usage of the device.

The returned type is a defaultdict, so there is no need to check if a key exists before accessing it.

exclude: name of chute whose device reservations should be excluded

**getInterfaceReservations**(*exclude=None*)
Get current set of interface reservations.

Returns an instance of InterfaceReservationSet.

exclude: name of chute whose interfaces should be excluded

**getReservations**(*update*)
Get device and resource reservations claimed by other users.

**getSubnetReservations**(*exclude=None*)
Get current set of subnet reservations.

Returns an instance of SubnetReservationSet.

exclude: name of chute whose reservations should be excluded

**paradrop.core.config.resource module**
**computeResourceAllocation**(*chutes*)
**getResourceAllocation**(*update*)

**paradrop.core.config.services module** Configure optional additional services such as telemetry.

**configure_telemetry**(*update*)

**paradrop.core.config.snap module**
**updateSnap**(*update*)

**paradrop.core.config.state module**
**removeAllChutes**(*update*)
**revertChute**(*update*)

**saveChute**(*update*)

**paradrop.core.config.uciutils module**
**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

Raises exception if an error occurs.

**paradrop.core.config.wifi module**
**getOSWirelessConfig**(*update*)
Read settings from networkInterfaces for wireless interfaces. Store wireless configuration settings in osWirelessConfig.
**setOSWirelessConfig**(*update*)
Write settings from osWirelessConfig out to UCI files.

**paradrop.core.config.zerotier module**

**configure**(*update*)

**getAddress**()

Return the zerotier address for this device or None if unavailable.

**get_auth_token**()

Return the zerotier auth token for accessing its API.

**get_networks**()

Get list of active ZeroTier networks.

**manage_network**(*nwid*, *action='join'*)

Join or leave a ZeroTier network.

nwid: ZeroTier network ID, e.g. "e5cd7a9e1c8a5e83" action: either "join" or "leave"

**Module contents**

### paradrop.core.container package

**Submodules**

**paradrop.core.container.chutecontainer module**

**class ChuteContainer**(*name*, *docker_url='unix://var/run/docker.sock'*)

Bases: object

Class for accessing information about a chute's container.

**getID**()

Look up the container ID as used by Docker.

**getIP**()

Look up the IP address assigned to the container.

**getPID**()

Look up the PID of the container, if running.

**getPortConfiguration**(*port*, *protocol='tcp'*)

Look up network port configuration. This tells us if a port in the host is bound to a port inside the container.

Returns a list, typically with zero or one elements.

Example:

[{ "HostIp": "0.0.0.0", "HostPort": "32768"

}]

**getStatus**()

Return the status of the container (running, exited, paused).

Returns "missing" if the chute does not exist.

**inspect**()

Return the full container status from Docker.

**isRunning**()

Check if container is running.

Returns True/False; returns False if the container does not exist.

**paradrop.core.container.dockerapi module** Functions associated with deploying and cleaning up docker containers.

**buildImage**(*update*)
Build the Docker image and monitor progress.

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

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

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

**call_in_netns**(*chute*, *env*, *command*, *onerror='raise'*, *pid=None*)
Call command within a chute's namespace.

command: should be a list of strings. onerror: should be "raise" or "ignore"

**call_retry**(*cmd*, *env*, *delay=3*, *tries=3*)

**cleanup_net_interfaces**(*chute*)
Cleanup special interfaces when bringing down a container.

This applies to monitor mode interfaces, which need to be renamed before they come back to the host network, e.g. "mon0" inside the container should be renamed to the appropriate "wlanX" before the container exits.

**getBridgeGateway**()
Look up the gateway IP address for the docker bridge network.

This is the docker0 IP address; it is the IP address of the host from the chute's perspective.

**getImageName**(*chute*)

**getPortList**(*chute*)
Get a list of ports to expose in the format expected by create_container.

Uses the port binding dictionary from the chute host_config section. The keys are expected to be integers or strings in one of the following formats: "port" or "port/protocol".

Example: port_bindings = {

"1111/udp": 1111, "2222": 2222

} getPortList returns [(1111, 'udp'), (2222, 'tcp')]

**prepare_environment**(*chute*)
Prepare environment variables for a chute container.

**prepare_port_bindings**(*chute*)

**removeAllContainers**(*update*)
Remove all containers on the system. This should only be used as part of a factory reset mechanism.

**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

**removeNewContainer**(*update*)
Remove the newly started container during abort sequence.

**removeNewImage**(*update*)
Remove the newly built image during abort sequence.

**removeOldContainer**(*update*)
Remove the docker container for the old version of a chute.

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

**Returns** None

**removeOldImage**(*update*)
Remove the image for the old version of the chute.

**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

**revertResourceAllocation**(*update*)

**setResourceAllocation**(*update*)

**setup_net_interfaces**(*chute*)
Link interfaces in the host to the internal interfaces in the Docker container.

The commands are based on the pipework script (https://github.com/jpetazzo/pipework).

**Parameters** **chute** – The chute object containing information about the chute.

**Returns** None

**startChute**(*update*)
Create a docker container based on the passed in update.

**startOldContainer**(*update*)
Create a docker container using the old version of the image.

**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.core.container.dockerfile module** This module generates a Dockerfile for use with light chutes.

**class Dockerfile**(*config*)
Bases: object

**getBytesIO**()
Geterate a Dockerfile and return as a BytesIO object.

**getString**()
Generate a Dockerfile as a multi-line string.

**isValid**()
Check if configuration is valid.

Returns a tuple (True/False, None or str).

**requiredFields = ['use', 'command']**

**writeFile**(*path*)
Generate Dockerfile and write to a file.

**paradrop.core.container.downloader module** This module downloads a package from a given URL using one of potentially many different methods. We currently support the github web API and simple HTTP(S). The github method is more developed and returns meta data about the project (the commit hash and message), but support for other methods, e.g. download a tar file that was uploaded to a web server, are not precluded.

Private downloads are supported with the HTTP Authorization header. For github, we need to use the github API to request a token to access the owner's private repository. That part is not implemented here.

**class Downloader**(*url*, *user=None*, *secret=None*, *repo_owner=None*, *repo_name=None*)
Bases: object

**download**()

**extract**()

**fetch**()
Download the project.

Returns the full path to the temporary directory containing the project and a dictionary containing meta data.

**meta**()

**class GithubDownloader**(*url*, *checkout='master'*, ***kwargs*)
Bases: *paradrop.core.container.downloader.Downloader*

**download**()

**meta**()
Return repository meta data as a dictionary.

**class WebDownloader**(*url*, *user=None*, *secret=None*, *repo_owner=None*, *repo_name=None*)
Bases: *paradrop.core.container.downloader.Downloader*

**download**()

**meta**()
Return repository meta data as a dictionary.

**downloader**(*url*, *user=None*, *secret=None*, ***kwargs*)
Return an appropriate Downloader for the given URL.

This should be used in a "with ... as ..." statement to perform cleanup on all exit cases.

Example: with downloader("https://github.com/...") as dl:

path, meta = dl.fetch() # do some work on the repo here

**paradrop.core.container.log_provider module** Provides messages from container logs (STDOUT and STDERR).

**class LogProvider**(*chutename*)
Bases: object

**attach**()
Start listening for log messages.

Log messages in the queue will appear like the following: {

'timestamp': '2017-01-30T15:46:23.009397536Z', 'message': 'Something happened'

}

**detach**()
Stop listening for log messages.

After this is called, no additional messages will be added to the queue.

**get_logs**()

**monitor_logs**(*chute_name*, *queue*, *tail=200*)
Iterate over log messages from a container and add them to the queue for consumption. This function will block and wait for new messages from the container. Use the queue to interface with async code.

tail: number of lines to retrieve from log history; the string "all" is also valid, but highly discouraged for performance reasons.

### Module contents

## paradrop.core.plan package

### Submodules

**paradrop.core.plan.executionplan module** This module contains the methods required to generate and act upon execution plans.

An execution plan is a set of operations that must be performed to update a Chute from some old state into the new state provided by the API server.

All plans that are generated are function pointers, as in no actual operations are performed during the generation process.

**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.core.plan.hostconfig module** This module generates update plans for a host configuration operation. It is separate from the modules that generate plans for chute operations because we only need to do a subset of the operations.

**generatePlans**(*update*)

**paradrop.core.plan.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.core.plan.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 a list of tuple 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.core.plan.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

**paradrop.core.plan.router module** This module generates update plans for router operations such as factory reset.

**generatePlans**(*update*)

**paradrop.core.plan.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.core.plan.snap module** This module generates update plans for a snap operation.

**generatePlans**(*update*)

**paradrop.core.plan.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

**paradrop.core.plan.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.core.plan.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

**Module contents**

### paradrop.core.system package

**Submodules**

**paradrop.core.system.system_info module** Get system information

**getDMI**()
Read hardware information from DMI.

This function attempts to read from known files in /sys/class/dmi/id/. If any are missing or an error occurs, those fields will be omitted from the result.

Returns: a dictionary with fields such as bios_version and product_serial.

**getOSVersion**()
Return a string identifying the host OS.

**getPackageVersion**(*name*)
Get a python package version.

Returns: a string or None

**paradrop.core.system.system_status module** Get system running status including CPU load, memory usage, network traffic.

**class SystemStatus**
Bases: object

**INCLUDED_PARTITIONS = set(['/writable', '/'])**

**classmethod getNetworkInfo**()

**classmethod getProcessInfo**(*pid*)

**getStatus**(*max_age=0.8*)
Get current system status.

max_age: maximum tolerable age of cached status information. Set to None to force a refresh regardless of cache age.

Returns a dictionary with fields 'cpu_load', 'mem', 'disk', and 'network'.

**classmethod getSystemInfo**()

**refreshCpuLoad**()

**refreshDiskInfo**()

**refreshMemoryInfo**()

**refreshNetworkTraffic**()

**Module contents**

### paradrop.core.update package

**Submodules**

**paradrop.core.update.update_fetcher module** Fetch new updates from the pdserver and apply the updates

**class UpdateFetcher**(*update_manager*)
Bases: object

**pull_update**(**args*, ***kwargs*)
Start updates by polling the server for the latest updates.

This is the only method that needs to be called from outside. The rest are triggered asynchronously.

Call chain: pull_update -> _updates_received -> _update_complete

_auto: Set to True when called by the scheduled LoopingCall.

**start_polling**(**args*, ***kwargs*)

**paradrop.core.update.update_manager module**

**class UpdateManager**(*reactor*)
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.

**add_update**(***update*)
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.

**assign_change_id**()
Get a unique change ID for an update.

This should be used to set the change_id field in an update object.

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

**find_change**(*change_id*)
Search active and queued changes for the requested change.

Returns an Update object or None.

**paradrop.core.update.update_object 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.core.update.update_object.UpdateObject*

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

**updateModuleList = [<module 'paradrop.core.plan.name' from '/home/docs/checkouts/readthedocs.org/user_builds/p**

**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...

**add_message_observer**(*observer*)

**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 plan 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.

**progress**(*message*)

**remove_message_observer**(*observer*)

**started**()
This function should be called when the updated object is dequeued and execution is about to begin.

Sends a notification to the pdserver if this is a tracked update.

**updateModuleList = []**

**class UpdateRouter**(*obj*)
Bases: *paradrop.core.update.update_object.UpdateObject*

Updates specifically tailored to router configuration.

**updateModuleList = [<module 'paradrop.core.plan.hostconfig' from '/home/docs/checkouts/readthedocs.org/user_bui**

**class UpdateSnap**(*obj*)
Bases: *paradrop.core.update.update_object.UpdateObject*

Updates specifically tailored to installing snaps.

**updateModuleList = [<module 'paradrop.core.plan.snap' from '/home/docs/checkouts/readthedocs.org/user_builds/pa**

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

**Module contents**

### Module contents

## paradrop.lib package

### Subpackages

#### paradrop.lib.misc package

### Submodules

#### paradrop.lib.misc.pdinstall module

**sendCommand**(*command*, *data*)

Send a command to the pdinstall service.

Commands: install - Install snaps from a file path or http(s) URL.

Required data fields: sources - List with at least one snap file path or URL. The snaps

are installed in order until one succeeds or all fail.

Returns True/False for success. Currently, we cannot check whether the call succeeded, only whether it was delived. A return value of False means we could not deliver the command to pdinstall.

#### paradrop.lib.misc.procmon module

The ProcessMonitor class ensures that a service is running and that its pid file is consistent.

This addresses an issue we have had with Docker on Ubuntu Snappy, where its pid file sometimes persists and prevents the service from starting.

**class ProcessMonitor**(*service*, *cmdstring=None*, *pidfile=None*, *action='restart'*)

Bases: object

**allowedActions = set(['reboot', 'restart'])**

**check**()

Check that the service is running and consistent with pid file(s).

Returns True or False.

**ensureReady**(*delay=5*, *tries=3*)

Look through checking and restarting the service until it is ready or the maximum number of tries has been reached.

delay: time delay (seconds) between retries. tries: maximum number of restart-wait-check cycles.

**restart**()

Restart the service.

#### paradrop.lib.misc.resopt module

Resource optimization functions.

**allocate**(*reservations*, *total=1.0*)

Allocate resources among slices with specified and unspecified reservations.

Returns a new list of values with the following properties: - Every value is >= the corresponding input value. - The result sums to *total*.

Examples: allocate([0.25, None, None]) -> [0.5, 0.25, 0.25] allocate([0.4, None, None]) -> [0.6, 0.2, 0.2] allocate([0.2, 0.2, 0.2]) -> [0.33, 0.33, 0.33] allocate([None, None, None]) -> [0.33, 0.33, 0.33] allocate([0.5, 0.5, 0.5]) -> ERROR

#### paradrop.lib.misc.snapd module

**class SnapdClient**(*logging=True*, *wait_async=False*)

Bases: object

Client for interacting with the snapd API to manage installed snaps.

**connect**(*plug_snap='paradrop-daemon'*, *plug=None*, *slot_snap='core'*, *slot=None*)

Connect an interface.

**get_change**(*change_id*)
Get the current status of a change.

**installSnap**(*snapName*)
Install a snap from the store.

**listSnaps**()
Get a list of installed snaps.

**updateSnap**(*snapName*, *data*)
Post an update to a snap.

Valid actions are: install, refresh, remove, revert, enable, disable.

Example: updateSnap("paradrop-daemon", {"action": "refresh"}

**paradrop.lib.misc.ssh_keys module**
**addAuthorizedKey**(*key*, *user='paradrop'*)
**getAuthorizedKeys**(*user='paradrop'*)

**writeAuthorizedKeys**(*keys*, *user='paradrop'*)

**Module contents**

### paradrop.lib.utils package

**Submodules**

**paradrop.lib.utils.addresses module**
**checkPhyExists**(*radioid*)
Check if this chute exists at all, a directory /sys/class/ieee80211/phyX must exist.
**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**(*ipaddr*, *netmask*)

**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.

**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.

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

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

**Returns:** Max quad dot IP string or None if error

**paradrop.lib.utils.datastruct module** Utilities for reading from data structures.

**getValue**(*struct*, *path*, *default=None*)
Read a value from the data structure.

Arguments: struct can comprise one or more levels of dicts and lists. path should be a string using dots to separate levels. default will be returned if the path cannot be traced.

Example: getValue({'a': [1, 2, 3]}, "a.1") -> 2 getValue({'a': [1, 2, 3]}, "a.3") -> None

**paradrop.lib.utils.pd_storage module**
**class PDStorage**(*filename*, *saveTimer*)
Bases: object

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.

**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.pdos module**
**basename**(*x*)
**copy**(*a*, *b*)

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

**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*)

**listdir**(*p*)

**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**(*path*, *suppressNotFound=False*)

**symlink**(*a*, *b*)

**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.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:**

  **{** ..., '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', '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.

**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='paradrop'*, *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.

**getLineParts**(*line*)
Split the UCI line into its whitespace-separated parts.

Returns a list of strings, with apostrophes removed.

**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*)

**singleConfigMatches**(*a*, *b*)

**stringify**(*a*)
Recursively convert all primitives in a data structure to strings.

**stringifyOptionValue**(*value*)
Convert option value from in-memory representation to a suitable string.

In particular, boolean values are converted to '0' or '1'.

**paradrop.lib.utils.uhttp module**

**class UHTTPConnection**(*path*)

Bases: httplib.HTTPConnection

Subclass of Python library HTTPConnection that uses a unix-domain socket.

Source: http://7bits.nl/blog/posts/http-on-unix-sockets-with-python

**connect**()

**Module contents**

### Module contents

## Submodules

## paradrop.main module

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

**class Nexus**(*update_fetcher*)

Bases: *paradrop.base.nexus.NexusBase*

**onStart**(**args*, ***kwargs*)

**onStop**()

**main**()

## paradrop.plan_demo module

This module is here purely to help with understanding the rather complex execution plan in Paradrop. Simply run it (python -m paradrop.plan_demo), and it will walk through all of the functions that make up a chute creation operation.

**loadPriorityMap**()

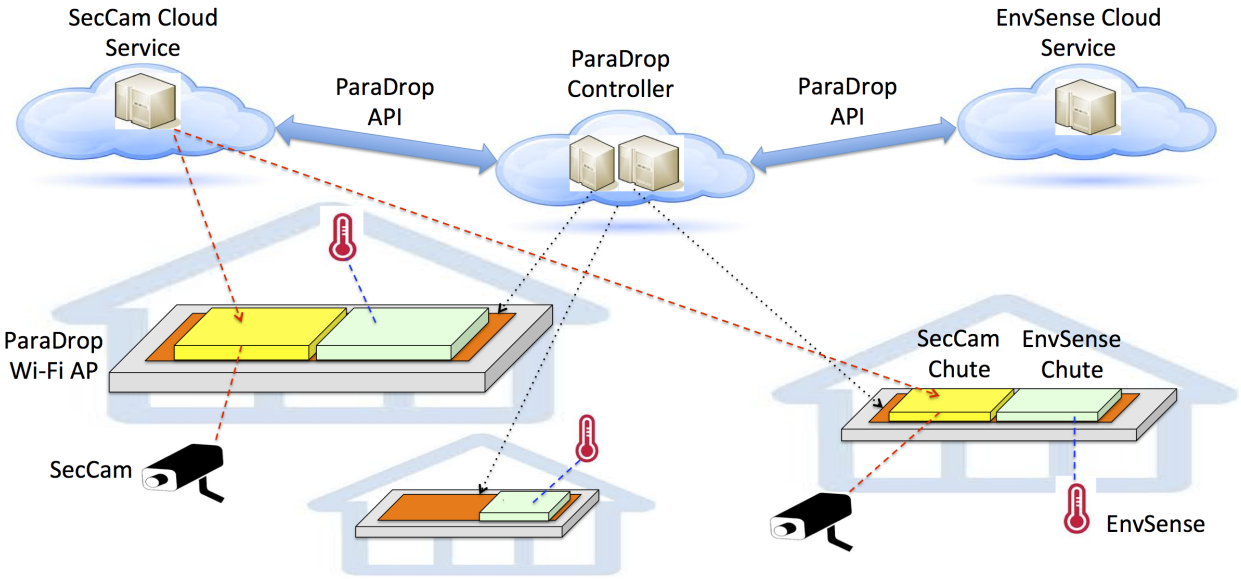
Make a map of priority values back to their names for reference.

These are defined as constant integer values in paradrop.backend.exc.plangraph. For example, for priority 9 (STRUCT_GET_SYSTEM_DEVICES), the dictionary produced by this function would contain the entry 9: "STRUCT_GET_SYSTEM_DEVICES".

## Module contents

# ParaDrop - Enabling Edge Computing at the Extreme Edge

ParaDrop is an open source edge computing platform developed by the WiNGS Lab at the University of Wisconsin-Madison. We built the ParaDrop platform with WiFi routers, so that we can "paradrop" services from the cloud to the extreme wireless edge - just one hop from user's mobile devices, data sources, and actuators of IoT applications. The name "ParaDrop" comes from the ability to "drop" supplies and resources ("services") into the network edge.




The above figure gives a high level overview of ParaDrop, including the ParaDrop platform and two example applications. With the ParaDrop API, third-party applications can deploy services into the network edge - the WiFi routers. More information about the design and evolution of ParaDrop can be found in the paper.

CHAPTER 13

# Getting Started

Please visit the Quick Start page for a quick introduction about how to use ParaDrop.

CHAPTER 14

# Where to go from here?

We have document about ParaDrop application development found under Developing Applications. If you are interested in working on the development of the ParaDrop platform (our github code) then check out: How to Contribute.

## /api

## p

## A



## B

## C

## D



## E



## F



## G

## H



## I

## J



## K



## L



## M



## N



## O

## P

## R

## S

## T



## U



## V



## W