Compoundpi Documentation

Release 0.4

Dave Hughes

January 30, 2017

Contents

This project provides a means for controlling multiple cameras attached to Raspberry Pis all of which are attached to the same subnet. Broadcast UDP packets are utilized to permit near-simultaneous triggering of all attached cameras.

CHAPTER 1

Links

- The code is licensed under the GPL v2 or above
- The source code can be obtained from GitHub, which also hosts the bug tracker
- The documentation (which includes installation and quick start examples) can be read on ReadTheDocs
- Packages can be downloaded from PyPI, although reading the installation instructions will probably be more useful

Table of Contents

2.1 Quick Start

By far the easiest method of configuring a fleet of Compound Pi servers is to get a single Pi running the *Compound Pi daemon* successfully, using an automatic network configuration, then clone its SD card for all the other Pis.

This quick start tutorial assumes you are using the Raspbian operating system on your Pi servers, and Ubuntu as your client.

2.1.1 Terminology

You may have noted above that we refer to the Pis as "servers" and the controlling computer as the "client". This may seem confusing, but there is a logic to it: each Pi is a server insofar as it sits there listening for orders from a network client. When it receives orders, it carries them out and sends back the results. This is akin to the way web servers sit on the Internet waiting for a browser to request pages from them. When a request comes along they look up, or generate the HTML response, and send it back to the browser.

The differences here are that Compound Pi operates over broadcast UDP rather than point-to-point TCP, and thus that it is limited to LAN operation.

Warning: You cannot (and should not attempt to) operate Compound Pi over the Internet; the Compound Pi server has almost no security features. It's intended to be a LAN-only daemon so don't open its port (5647 by default) to the Internet at large!

2.1.2 Hardware Selection

Before doing anything it's worth thinking about what hardware to use in your setup. Firstly, there's the selection of Pi to use. The primary concern here is RAM size. Compound Pi uses each Pi's memory to store captured images to avoid dealing with any lengthy delays writing to SD cards (this isn't simply a matter of slow SD cards, but avoiding periodic flushes of the Linux disk cache which can severely impact the timing of shots).

To this end, the more RAM in your Pi, the better. Compound Pi is capable of running on a model A or A+ (256Mb of RAM) but after the GPU has taken its share (128Mb) and the OS and Compound Pi server have grabbed theirs, there's typically less than 100Mb left for data storage. The model B or B+ (512Mb of RAM) is a better selection typically providing over 300Mb of temporary data storage. However, the Pi 2 model B (1Gb of RAM) is the obviously the ultimate choice as it typically has 800Mb or more of available memory for data storage, and the faster processor doesn't hurt either.

The next important selection is the network between your client and Pi servers. Compound Pi can run over WiFi (in fact, this was the first configuration it was tested in) but there are numerous reasons why WiFi is sub-optimal:

- WiFi has much worse ping times than Ethernet. Ping time is important to obtaining well synchronized shots with Compound Pi.

- WiFi is more complex to configure and debug when it goes wrong. A standard NOOBS installation will work automatically over Ethernet with DHCP, but WiFi typically requires association configuration and in a headless setup it can be extremely annoying to debug.
- Many WiFi adapters switch themselves off after idle periods to conserve power. This is a perfectly reasonable thing to do when attached to a laptop running on batteries, but it's useless in the context of a server which has to listen constantly for requests from the client.

While WiFi may be tempting because of the lack of wires needed between the client and all the Pi servers, it is certainly not the optimal setup for running Compound Pi.

Although it costs significantly more, the ultimate Compound Pi setup would involve power over Ethernet (POE), allowing a single cable to run to each Pi carrying both data and power. Given you have to run one cable anyway for power, this minimizes the number of cables, while providing the best connectivity. The only downside is the cost of a POE capable switch (typically >$100) and a POE HAT for each Pi to split out the power from the data.

2.1.3 Client Installation

Ensure your Ubuntu client machine is connected to the same network as your Pis (whether by Ethernet or Wifi doesn't matter). Then, execute the following to install the client and an NTP daemon:

```
$ sudo add-apt-repository ppa:waveform/ppa
$ sudo apt-get update
$ sudo apt-get install compoundpi-client ntp
```

The NTP daemon will most likely be installed to synchronize with an NTP pool on the Internet (e.g. `pool.ntp.org`). This is fine, but check that it's working with the following command line:

```
$ ntpq -p
     remote           refid      st t when poll reach   delay   offset  jitter
==============================================================================
*aaaaaaa.aaaaaaa nn.nnn.nnn.nnn   3 u  109 1024  377    4.639   -2.101  21.233
```

2.1.4 Server Network Configuration

On the Pi you intend to clone, configure networking to use DHCP to automatically obtain an IP address. Edit the `/etc/network/interfaces` file and ensure that it looks similar to the following:

```
auto lo

iface lo inet loopback
iface eth0 inet dhcp

allow-hotplug wlan0
iface wlan0 inet manual
wpa-roam /etc/wpa_supplicant/wpa_supplicant.conf
iface default inet dhcp
```

This configuration should ensure that the first Ethernet and/or WiFi interfaces will pick up an address automatically from the local DHCP server. If you are using WiFi, complete the WiFi configuration by editing the `/etc/wpa_supplicant/wpa_supplicant.conf` file to look something like the following:

```
ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=netdev
update_config=1

network={
        ssid="my_wireless_ssid"
        psk="my_wireless_password"
        proto=RSN
        key_mgmt=WPA-PSK
        pairwise=CCMP
```

```
        auth_alg=OPEN
}
```

2.1.5 Server Installation

Execute the following command to install the Compound Pi server package and the NTP daemon (the latter is required for time-synchronized image capture):

```
$ sudo apt-get install compoundpi-server ntp
```

This should pull in all necessary dependencies, and automatically install an init-script which will start the Compound Pi daemon on boot-up. Test this by rebooting the Pi with a camera module attached. You should see the camera module's LED light up when the daemon starts. If it doesn't, the most likely culprit is the camera: try running **raspistill**, ensure you've activated the camera with **sudo raspi-config**, and ensure the CSI cable is inserted correctly. You can control the Compound Pi daemon as you would any other system daemon:

```
$ sudo service cpid stop
$ sudo service cpid start
$ sudo service cpid restart
```

Ideally, you want all your Pi servers to sync with the NTP time server you set up on your client. Edit the `/etc/ntp.conf` file and replace the `server` lines with the IP address of your client (ideally you should configure your router to give your client a fixed address):

```
...
#server 0.debian.pool.ntp.org iburst
#server 1.debian.pool.ntp.org iburst
#server 2.debian.pool.ntp.org iburst
#server 3.debian.pool.ntp.org iburst
server 192.168.1.2
...
```

Restart the NTP daemon to use the new configuration:

```
$ sudo service ntp restart
```

2.1.6 Clone the SD Card

Once you've got a Pi running the Compound Pi daemon successfully, shut it down and place its SD card in any Linux machine with an SD card reader. Unmount any partitions that auto-mount, then figure out which device node represents the SD card. For example, the following would tell you that the SD card is `sdd`:

```
$ dmesg | tail | grep "Attached SCSI removable disk"
[    3.428459] sd 8:0:0:0: [sdd] Attached SCSI removable disk
```

Clone the SD card into a disk file:

```
$ sudo dd if=/dev/sdd of=server.img bs=1M
```

This will take some considerable time to finish. Once it has done so, eject the source SD card and insert the target one in its place. Remember to unmount any partitions which auto-mount, then execute the reverse command:

```
$ sudo dd if=server.img of=/dev/sdd bs=1M
```

Repeat this last step for all remaining target cards. Finally, install the SD cards in your set of Pi servers and boot them all to ensure their camera modules activate.

Warning: Ensure your target SD cards are the same size or larger than the source SD card. If they are larger, they will still appear the same size as the source after cloning because you the cloning also duplicates the partition table of the smaller device.

2.1.7 Testing the Servers

Back on the Ubuntu client machine, execute *cpi* to run the client. You will be presented with a command line like the following:

```
CompoundPi Client version 0.4
Type "help" for more information, or "find" to locate Pi servers
cpi>
```

Firstly, ensure that the network configuration is correct. The *config* command can be used to print the current configuration:

```
cpi> config
Setting             Value
------------------- --------------
network             192.168.0.0/16
port                5647
bind                0.0.0.0:5647
timeout             15
capture_delay       0.0
capture_quality     85
capture_count       1
video_port          False
record_delay        0.0
record_format       h264
record_quality      0
record_bitrate      17000000
record_motion       False
record_intra_period 30
time_delta          0.25
output              /tmp
warnings            False
```

Assuming we're using a typical home router which gives out addresses in the 192.168.1.x network, this is incorrect. In order for broadcasts to work, the network *must* have the correct definition - it's no good having a superset configured (192.168.0.0/16 is a superset of 192.168.1.0/24). See IPv4 subnetting for more information about subnet configuration.

To correct the network definition, use the *set* command:

```
cpi> set network 192.168.1.0/24
cpi> config
Setting             Value
------------------- --------------
network             192.168.1.0/24
port                5647
bind                0.0.0.0:5647
timeout             15
capture_delay       0.0
capture_quality     85
capture_count       1
video_port          False
record_delay        0.0
record_format       h264
record_quality      0
record_bitrate      17000000
record_motion       False
record_intra_period 30
time_delta          0.25
output              /tmp
warnings            False
```

To make permanent configuration changes, simply place them in a file named `~/.cpi.ini` like so:

```
[cpi]
network=192.168.1.0/24
timeout=10
output=~/Pictures
```

With the network configured correctly, you can now use *find* to locate your servers. If you run *find* on its own it will send out a broadcast ping and wait for a fixed number of seconds for servers to respond. If you know exactly how many servers you have, specify a number with the *find* command and it will warn you if it doesn't find that many servers (it will also finish faster if it does find the expected number of Pis):

```
cpi> find 2
Found 2 servers
```

You can query the status of your servers with the *status* command which will give you the basics for the camera configuration, the time according to the server, and the number of images currently stored in memory on the server. If you only want to query a specific set of servers you can give their addresses as a parameter:

```
cpi> status 192.168.1.154
Address       Mode        AGC             AWB             Exp             Meter   Flip Clock
------------- ----------- --------------- --------------- --------------- ------- ---- -------------
192.168.1.154 1280x800@30 auto (1.0,1.0) auto (1.6,1.3) auto (28.48ms) average none 0:00:00
```

If any major discrepancies are detected (resolution, framerate, timestamp, etc.), the status command should notify you of them. The maximum discrepancy permitted in the timestamp is configured with the `time_delta` configuration setting.

To shoot an image, use the *capture* command:

```
cpi> capture
```

Finally, to download the captured images from all Pis, simply use the *download* command:

```
cpi> download
Downloaded image 0 from 192.168.1.154
Downloaded image 0 from 192.168.1.168
```

You can use the *config* and *set* commands to configure capture options, the download target directory, and so on.

Since version 0.3 a GUI client is also provided. The basic operations of the GUI client are essentially the same as the command line client, the only major difference being that download is performed automatically after capture. You can start the GUI client with the cpigui command.

2.1.8 Generating video

Once you have images captured from your array of Pi servers, you may wish to convert them into video (e.g. for bullet-time effects and such like). The ordering of captured images is currently relatively tricky. However, once you have your images in an order that you like you can use the following ffmpeg command line to convert the series of JPEGs into an MP4 with H.264 encoding:

```
ffmpeg -y -f image2 -i frame%03d.jpg -r 24 -vcodec libx264 -profile high -preset slow output.mp4
```

The above command line assumes that your images are all named something like `frame001.jpg` or `frame027.jpg` and that they are in advancing numerical order. It also assumes that you wish the output to be called `output.mp4`. x264 compression is quite computationally intensive, so this is something you want to do on a platform with a fair amount of power (like a full PC).

2.1.9 Troubleshooting

Compound Pi provides some crude but effective tools for debugging problems. The first is simply that the daemon activates the camera by default. If you see a Pi server without the camera LED lit after boot-up, you know the daemon has failed to start for some reason.

The *identify* command is the main debugging tool provided by Compound Pi. If specified without any further parameters it will cause all discovered Pi servers to blink their camera LED for 5 seconds. Thus, if you run this command immediately after *find* you can quickly locate any Pi servers that were not discovered (typically this is due to misconfiguration of the network).

If *identify* is specified with one or more addresses, it will blink the LED on the specified Pi servers. This can be used to quickly figure out which address corresponds to which Pi (useful when dynamic addressing is used).

2.2 Server Installation

The server component of Compound Pi can only be installed on the Raspberry Pi architecture. On Raspbian, the following command can be used to install the server package:

```
$ sudo apt-get install compoundpi-server
```

Warning: The Raspbian package will automatically install the *cpid* daemon in the boot sequence. This will make the camera inaccessible to other processes unless the daemon is manually stopped or prevented from starting.

On other platforms, the package can be installed from PyPI. First, ensure that you have the `pip` command installed (this is in the `python-pip` package on Debian and RedHat based distros). Use this to install compound pi, specifying the `server` option to pull in all dependencies required by the server component:

```
$ sudo pip install "compoundpi[server]"
```

Warning: The PyPI package does not include init-scripts (because it can't). You will need to write these for your platform manually if you wish the daemon to start automatically on boot-up.

2.3 Client Installation

The client component of Compound Pi can be installed on any machine with Python available. On Ubuntu, the Waveform PPA can be used for simple installation:

```
$ sudo add-apt-repository ppa:waveform/ppa
$ sudo apt-get update
$ sudo apt-get install compoundpi-client
```

On Raspbian (assuming you want to use a Raspberry Pi as a client), use the same procedure as for Ubuntu (above) but omit the `add-apt-repository` step. Be aware that the GUI client is untested under Raspbian.

On other platforms, the package can be installed from PyPI. First, ensure that you have the `pip` command installed (this is in the `python-pip` package on Debian and RedHat based distros). Use this to install compound pi, specifying the `client` option to pull in all dependencies required by the client component:

```
$ sudo pip install "compoundpi[client]"
```

Warning: Currently, the version of client and server must match exactly. The client will not work with a different version server (either older or newer).

2.4 cpi

This is the Compound Pi client application which provides a command line interface through which you can query and interact with any Pi's running the *Compound Pi daemon* on your configured subnet. Use the *help* command within the application for information on the available commands.

The application can be configured via command line switches, a configuration file (defaults to `/etc/cpi.ini`, `/usr/local/etc/cpi.ini`, or `~/.cpid.ini`), or through the interactive command line itself.

2.4.1 Synopsis

```
cpi [-h] [--version] [-c CONFIG] [-q] [-v] [-l FILE] [-P] [-o PATH]
    [-n NETWORK] [-p PORT] [-b ADDRESS:PORT] [-t SECS]
    [--capture-delay SECS] [--capture-count NUM] [--video-port]
```

2.4.2 Description

-h, --help
show this help message and exit

--version
show program's version number and exit

-c CONFIG, **--config** CONFIG
specify a configuration file to load

-q, --quiet
produce less console output

-v, --verbose
produce more console output

-l FILE, **--log-file** FILE
log messages to the specified file

-P, --pdb
run under PDB (debug mode)

-o PATH, **--output** PATH
specifies the directory that downloaded images will be written to (default: `/tmp`)

-n NETWORK, **--network** NETWORK
specifies the network that the servers belong to (default: 192.168.0.0/16)

-p PORT, **--port** PORT
specifies the port that the servers will be listening on (default: 5647)

-b ADDRESS:PORT, **--bind** ADDRESS:PORT
specifies the address and port that the client listens on for downloads (default: 0.0.0.0:5647)

-t SECS, **--timeout** SECS
specifies the timeout (in seconds) for network transactions (default: 5)

--capture-delay SECS
specifies the delay (in seconds) used to synchronize captures. This must be less than the network timeout (default: 0)

--capture-count NUM
specifies the number of consecutive pictures to capture when requested (default: 1)

--video-port
if specified, use the camera's video port for rapid capture

2.4.3 Usage

The first command in a Compound Pi session is usually *find* to locate the servers on the specified subnet. If you know the number of servers available, specify it as an argument to the *find* command which will cause the

command to return quicker in the case that all servers are found, or to warn you if less than the expected number are located.

The *status* command can be used to check that all servers have an equivalent camera configuration, and that time sync is reasonable.

The *capture* command is used to cause all located servers to capture an image. After capturing, use the *download* command to transfer all captured images to the client.

Finally, the *help* command can be used to query the available commands, and to obtain help on an individual command.

2.5 cpid

This is the server daemon for the Compound Pi application. Starting the application with no arguments starts the server in the foreground. The server can be configured through command line arguments or a configuration file (which defaults to `/etc/cpid.ini`, `/usr/local/etc/cpid.ini`, or `~/.cpid.ini`).

2.5.1 Synopsis

```
cpid [-h] [--version] [-c CONFIG] [-q] [-v] [-l FILE] [-P] [-b ADDRESS]
     [-p PORT] [-d] [-u UID] [-g GID] [--pidfile FILE]
```

2.5.2 Description

-h, --help
show this help message and exit

--version
show program's version number and exit

-c CONFIG, **--config** CONFIG
specify a configuration file to load

-q, --quiet
produce less console output

-v, --verbose
produce more console output

-l FILE, **--log-file** FILE
log messages to the specified file

-P, --pdb
run under PDB (debug mode)

-b ADDRESS, **--bind** ADDRESS
specifies the address to listen on for packets (default: 0.0.0.0)

-p PORT, **--port** PORT
specifies the UDP port for the server to listen on (default: 5647)

-d, --daemon
if specified, start as a background daemon

-u UID, **--user** UID
specifies the user that the daemon should run as. Defaults to the effective user (typically root)

-g GID, **--group** GID
specifies the group that the daemon should run as. Defaults to the effective group (typically root)

--pidfile FILE
specifies the location of the pid lock file

2.5.3 Usage

The Compound Pi server is typically started at boot time by the init service. The Raspbian package includes an init script for this purpose. Users on other platforms will need to write their own init script.

When the server starts successfully it will initialize the camera and hold it open. This will prevent other applications from using the camera but also makes it easy to see that the server has started as the camera's LED will be lit (this is useful as Compound Pi servers are typically headless).

Note: If you explicitly set a user and/or group for the daemon (with the *cpid -u* and *cpid -g* options), be aware that using the Pi's camera typically requires membership of the `video` group. Furthermore, the specified user and group must have the ability to create and remove the pid lock file.

2.6 Client Commands

Each section below documents one of the commands available in the Compound Pi command line client. Many commands accept an address or list of addresses. Addresses must be specified in dotted-decimal format (no hostnames). Inclusive ranges of addresses are specified by two dash-separated addresses. Lists of addresses, or ranges of addresses are specified by comma-separating each list item.

The following table demonstrates various examples of this syntax:

Syntax	Expands To
`192.168.0.1`	192.168.0.1
`192.168.0.1-192.168.0.5`	192.168.0.1 192.168.0.2 192.168.0.3 192.168.0.4 192.168.0.5
`192.168.0.1,192.168.0.3`	192.168.0.1 192.168.0.3
`192.168.0.1,192.168.0.3-192.168.0.5`	192.168.0.1 192.168.0.3 192.168.0.4 192.168.0.5
`192.168.0.1-192.168.0.3,192.168.0.5`	192.168.0.1 192.168.0.2 192.168.0.3 192.168.0.5

It is also worth noting that if readline is installed (which it is on almost any modern Unix platform), the command line supports `Tab`-completion for commands and most parameters, including defined server addresses.

2.6.1 add

Syntax: add *addresses*

The *add* command is used to manually define the set of Pi servers to communicate with. Addresses can be specified individually, as a dash-separated range, or a comma-separated list of ranges and addresses.

See also: *find*, *remove*, *servers*.

```
cpi> add 192.168.0.1
cpi> add 192.168.0.1-192.168.0.10
cpi> add 192.168.0.1,192.168.0.5-192.168.0.10
```

2.6.2 agc

Syntax: agc *mode [addresses]*

The *agc* command is used to set the AGC mode of the camera on all or some of the defined servers. The mode can be one of the following:

- antishake
- auto
- backlight
- beach
- fireworks
- fixedfps
- night
- nightpreview
- off
- snow
- sports
- spotlight
- verylong

If 'off' is specified, the current sensor gains of the camera will be fixed at their present values (unfortunately there is no way at the moment to manually specify the gain values).

If no address is specified then all currently defined servers will be targetted. Multiple addresses can be specified with dash-separated ranges, comma-separated lists, or any combination of the two.

See also: *status*, *awb*, *exposure*, *metering*.

```
cpi> agc auto
cpi> agc backlight 192.168.0.1
cpi> agc antishake 192.168.0.1-192.168.0.10
cpi> agc off
cpi> agc off 192.168.0.1
```

2.6.3 awb

Syntax: awb (*mode* | *red_gain blue_gain*) *[addresses]*

The *awb* command is used to set the AWB mode of the camera on all or some of the defined servers. The mode can be one of the following:

- auto
- cloudy
- flash
- fluorescent
- horizon
- incandescent
- shade
- sunlight
- tungsten

Alternatively you can specify the red and blue gains of the camera manually as two floating point values. Valid gains for each channel are between 0.0 and 8.0. Typical values are between 1.0 and 2.0 (for most scenes, red gain slightly exceeds blue gain, e.g. 1.6 and 1.2 respectively).

If no address is specified then all currently defined servers will be targetted. Multiple addresses can be specified with dash-separated ranges, comma-separated lists, or any combination of the two.

See also: *status*, *exposure*, *metering*.

```
cpi> awb auto
cpi> awb 1.5 1.3
cpi> awb fluorescent 192.168.0.1
cpi> awb 1.7 1.0 192.168.0.10
cpi> awb sunlight 192.168.0.1-192.168.0.10
```

2.6.4 brightness

Syntax: brightness *value [addresses]*

The *brightness* command is used to adjust the brightness level on all or some of the defined servers. Brightness is specified as an integer number between 0 and 100 (default 50).

If no address is specified then all currently defined servers will be targetted. Multiple addresses can be specified with dash-separated ranges, comma-separated lists, or any combination of the two.

See also: *contrast*, *saturation*, *ev*.

```
cpi> brightness 50
cpi> brightness 75 192.168.0.1
```

2.6.5 capture

Syntax: capture *[addresses]*

The *capture* command causes the servers to capture an image. Note that this does not cause the captured images to be sent to the client. See the *download* command for more information.

If no addresses are specified, a broadcast message to all defined servers will be used in which case the timestamp of the captured images are likely to be extremely close together. If addresses are specified, unicast messages will be sent to each server in turn. While this is still reasonably quick there will be a measurable difference between the timestamps of the last and first captures.

See also: *record*, *download*, *clear*.

```
cpi> capture
cpi> capture 192.168.0.1
cpi> capture 192.168.0.50-192.168.0.53
```

2.6.6 clear

Syntax: clear *[addresses]*

The *clear* command can be used to clear the in-memory image store on the specified Pi servers (or all Pi servers if no address is given). The *download* command automatically clears the image store after successful transfers so this command is only useful in the case that the operator wants to discard images without first downloading them.

See also: *download*, *capture*.

```
cpi> clear
cpi> clear 192.168.0.1-192.168.0.10
```

2.6.7 config

Syntax: config

The *config* command is used to display the current client configuration. Use the related *set* command to alter the configuration.

See also: *set*.

```
cpi> config
```

2.6.8 contrast

Syntax: contrast *value [addresses]*

The *contrast* command is used to adjust the contrast level on all or some of the defined servers. Contrast is specified as an integer number between -100 and 100 (default 0).

If no address is specified then all currently defined servers will be targetted. Multiple addresses can be specified with dash-separated ranges, comma-separated lists, or any combination of the two.

See also: *brightness*, *saturation*, *ev*.

```
cpi> contrast 0
cpi> contrast -50 192.168.0.1
```

2.6.9 denoise

Syntax: denoise *value [addresses]*

The *denoise* command is used to set whether the camera's software denoise algorithm is active when capturing. The follow values can be specified:

- on
- off

If no address is specified then all currently defined servers will be targetted. Multiple addresses can be specified with dash-separated ranges, comma-separated lists, or any combination of the two.

See also: *status*.

```
cpi> denoise off
cpi> denoise on 192.168.0.3
```

2.6.10 download

Syntax: download *[addresses]*

The *download* command causes each server to send its captured images to the client. Servers are contacted consecutively to avoid saturating the network bandwidth. Once images are successfully downloaded from a server, they are wiped from the server.

See also: *capture*, *clear*.

```
cpi> download
cpi> download 192.168.0.1
```

2.6.11 ev

Syntax: ev *value [addresses]*

The *ev* command is used to adjust the exposure compensation (EV) level on all or some of the defined servers. Exposure compensation is specified as an integer number between -24 and 24 where each increment represents 1/6th of a stop. Hence, 12 indicates that camera should overexpose by 2 stops. The default EV is 0.

If no address is specified then all currently defined servers will be targetted. Multiple addresses can be specified with dash-separated ranges, comma-separated lists, or any combination of the two.

See also: *brightness*, *contrast*, *saturation*.

```
cpi> ev 0
cpi> ev 6 192.168.0.1
```

2.6.12 exit

Syntax: exit|quit

The *exit* command is used to terminate the application. You can also use the standard UNIX `Ctrl+D` end of file sequence to quit.

2.6.13 exposure

Syntax: exposure (auto | *speed*) *[addresses]*

The *exposure* command is used to set the exposure mode of the camera on all or some of the defined servers. The mode can be 'auto' or a speed measured in ms. Please note that exposure speed is limited by framerate.

If no address is specified then all currently defined servers will be targetted. Multiple addresses can be specified with dash-separated ranges, comma-separated lists, or any combination of the two.

See also: *status*, *awb*, *metering*.

```
cpi> exposure auto
cpi> exposure 30 192.168.0.1
cpi> exposure auto 192.168.0.1-192.168.0.10
```

2.6.14 find

Syntax: find *[count]*

The *find* command is typically the first command used in a client session to locate all Pis on the configured subnet. If a count is specified, the command will display an error if the expected number of Pis is not located.

See also: *add*, *remove*, *servers*, *identify*.

```
cpi> find
cpi> find 20
```

2.6.15 flip

Syntax: flip *value [addresses]*

The *flip* command is used to set the picture orientation on all or some of the defined servers. The following values can be specified:

- none
- horizontal
- vertical
- both

If no address is specified then all currently defined servers will be targetted. Multiple addresses can be specified with dash-separated ranges, comma-separated lists, or any combination of the two.

See also: *status*.

```
cpi> flip none
cpi> flip vertical 192.168.0.1
cpi> flip both 192.168.0.1-192.168.0.10
```

2.6.16 framerate

Syntax: framerate *rate [addresses]*

The *framerate* command is used to set the capture framerate of the camera on all or some of the defined servers. The rate can be specified as an integer, a floating-point number, or as a fractional value. The framerate of the camera influences the capture mode that the camera uses. See the camera hardware chapter of the picamera documentation for more information.

If no address is specified then all currently defined servers will be targetted. Multiple addresses can be specified with dash-separated ranges, comma-separated lists, or any combination of the two.

See also: *status*, *resolution*.

```
cpi> framerate 30
cpi> framerate 90 192.168.0.1
cpi> framerate 15 192.168.0.1-192.168.0.10
```

2.6.17 help

Syntax: help *[command]*

The 'help' command is used to display the help text for a command or, if no command is specified, it presents a list of all available commands along with a brief description of each.

2.6.18 identify

Syntax: identify *[addresses]*

The *identify* command can be used to locate a specific Pi server (or servers) by their address. It sends a command causing the camera's LED to blink on and off for 5 seconds. If no addresses are specified, the command will be sent to all defined servers (this can be useful after the *find* command to determine whether any Pi's failed to respond due to network issues).

See also: *find*.

```
cpi> identify
cpi> identify 192.168.0.1
cpi> identify 192.168.0.3-192.168.0.5
```

2.6.19 iso

Syntax: iso *value [addresses]*

The *iso* command is used to set the emulated ISO value of the camera on all or some of the defined servers. The value can be specified as an integer number between 0 and 1600, or `auto` which leaves the camera to determine the optimal ISO value.

If no address is specified then all currently defined servers will be targetted. Multiple addresses can be specified with dash-separated ranges, comma-separated lists, or any combination of the two.

See also: *status*, *exposure*.

```
cpi> iso auto
cpi> iso 100 192.168.0.1
cpi> iso 800 192.168.0.1-192.168.0.10
```

2.6.20 metering

Syntax: metering *mode [addresses]*

The *metering* command is used to set the metering mode of the camera on all or some of the defined servers. The mode can be one of the following:

- average
- backlit
- matrix
- spot

If no address is specified then all currently defined servers will be targetted. Multiple addresses can be specified with dash-separated ranges, comma-separated lists, or any combination of the two.

See also: *status*, *awb*, *exposure*.

```
cpi> metering average
cpi> metering spot 192.168.0.1
cpi> metering backlit 192.168.0.1-192.168.0.10
```

2.6.21 move

Syntax: move *address* (top|bottom|to *index*|(above|below) *address*)

The *move* command is used to move a server to another position within the server list. The first address specified is moved to the position described by the subsequent parameters. The `top`, `bottom`, and `to` arguments specify absolute positions. Alternatively, `above` and `below` can be used to specify a position relative to another address.

See also: *add*, *remove*, *sort*, *servers*.

```
cpi> move 192.168.0.1 top
cpi> move 192.168.0.2 below 192.168.0.1
cpi> move 192.168.0.3 to 2
```

2.6.22 quit

Syntax: exit|quit

The *exit* command is used to terminate the application. You can also use the standard UNIX `Ctrl+D` end of file sequence to quit.

2.6.23 record

Syntax: record *length [addresses]*

The *record* command causes the servers to record video. Note that this does not cause the recorded video to be sent to the client. See the *download* command for more information. The length of time to record for is specified as a number of seconds.

If no addresses are specified, a broadcast message to all defined servers will be used in which case the timestamp of the recorded video are likely to be extremely close together. If addresses are specified, unicast messages will be sent to each server in turn. While this is still reasonably quick there will be a measurable difference between the timestamps of the last and first recordings.

See also: *capture*, *download*, *clear*.

```
cpi> record 5
cpi> record 10 192.168.0.1
cpi> record 2.5 192.168.0.50-192.168.0.53
```

2.6.24 remove

Syntax: remove *addresses*

The *remove* command is used to remove addresses from the set of Pi servers to communicate with. Addresses can be specified individually, as a dash-separated range, or a comma-separated list of ranges and addresses.

See also: *add*, *find*, *servers*.

```
cpi> remove 192.168.0.1
cpi> remove 192.168.0.1-192.168.0.10
cpi> remove 192.168.0.1,192.168.0.5-192.168.0.10
```

2.6.25 resolution

Syntax: resolution *width x height [addresses]*

The *resolution* command is used to set the capture resolution of the camera on all or some of the defined servers. The resolution of the camera influences the capture mode that the camera uses. See the camera hardware chapter of the picamera documentation for more information.

If no address is specified then all currently defined servers will be targetted. Multiple addresses can be specified with dash-separated ranges, comma-separated lists, or any combination of the two.

See also: *status*, *framerate*.

```
cpi> resolution 640x480
cpi> resolution 1280x720 192.168.0.54
cpi> resolution 1280x720 192.168.0.1,192.168.0.3
```

2.6.26 saturation

Syntax: saturation *value [addresses]*

The *saturation* command is used to adjust the saturation level on all or some of the defined servers. Saturation is specified as an integer number between -100 and 100 (default 0).

If no address is specified then all currently defined servers will be targetted. Multiple addresses can be specified with dash-separated ranges, comma-separated lists, or any combination of the two.

See also: *brightness*, *contrast*, *ev*.

```
cpi> saturation 0
cpi> saturation -50 192.168.0.1
```

2.6.27 servers

Syntax: servers

The *servers* command is used to list the set of servers that the client expects to communicate with. The content of the list can be manipulated with the *find*, *add*, and *remove* commands.

See also: *find*, *add*, *remove*, *move*, *sort*.

```
cpi> servers
```

2.6.28 set

Syntax: set *name value*

The *set* command is used to alter the value of a client configuration variable. Use the related *config* command to view the current configuration.

See also: *config*.

```
cpi> set timeout 10
cpi> set output ~/Pictures/
cpi> set capture_count 5
```

2.6.29 sort

Syntax: sort *[reverse]*

The *sort* command is used to sort the list of defined servers numerically forwards or, if `reverse` is specified, backwards.

See also: *add*, *remove*, *move*, *find*.

```
cpi> sort
cpi> sort reverse
```

2.6.30 status

Syntax: status *[addresses]*

The *status* command is used to retrieve configuration information from servers. If no addresses are specified, then all defined servers will be queried.

See also: *resolution*, *framerate*.

```
cpi> status
```

2.7 Building Batch Clients

While the command line (*cpi*) and GUI (cpigui) clients lend themselves to interactive use, neither is suited for batch use. Thankfully, the logic for communicating with Compound Pi camera servers is split out into its own class (*CompoundPiClient*) which is used by both clients. The class is relatively simple to use, and also lends itself to construction of batch scripts for controlling Compound Pi camera servers.

The following sections document the API of the class, and provide several examples of batch scripts.

2.7.1 CompoundPiClient

class compoundpi.client.**CompoundPiClient**(*progress=None*)

Implements a network client for Compound Pi servers.

The optional *progress* parameter provides an object which will be notified of long client operations. When the client begins a long operation it will call the `start` method of the object with a single parameter indicating the number of expected operations to complete. As the operation progresses, the object's `update` method will be called with a parameter indicating the current operation (the `update` method may be called multiple times with the same number, but it will never decrease within the span of one operation, and it will never exceed the count passed to `start`). To terminate a long operation prematurely, `raise` an exception in the `update` method. Finally, the object's `finish` routine will be called with no parameters (if the `start` method is called, the `finish` method is guaranteed to be called).

Before controlling any Compound Pi servers, the client must either be told the addresses of the servers, or discover them via broadcast. The *servers* attribute is the list of available servers. Servers can be defined

manually, or discovered by broadcast. See the *CompoundPiServerList* documentation for further information.

Various methods are provided for configuring and controlling the cameras on the Compound Pi servers (*resolution()*, *framerate()*, *exposure()*, *capture()*, etc). Each method optionally accepts a set of addresses to operate on. If omitted, the command is applied to all servers that the client knows about (via a broadcast packet).

The one exception to this is the *download()* method for retrieving captured images. For the sake of efficiency this is expected to operate against one server at a time, so the *address* parameter is mandatory. The class listens on port 5647 on all available interfaces for download transmissions. If this is incorrect (or if you wish to limit the interfaces that the client listens on), adjust the *bind* attribute.

When you are finished with the client, you must call the *close()* method which shuts down the listening socket and server thread. Failure to do so will likely cause your application or script to hang (the server thread is deliberately not marked as a daemon thread, so your script will not terminate while it is still active). For example:

```
from compoundpi.client import CompoundPiClient

client = CompoundPiClient()
try:
    client.servers.find(10)
    client.capture()
finally:
    client.close()
```

The client class can be used as a context handler to ensure this happens implicitly:

```
from compoundpi.client import CompoundPiClient

with CompoundPiClient() as client:
    client.servers.find(10)
    client.capture()
```

agc(*mode*, *addresses=None*)

Called to change the automatic gain control on the servers at the specified *addresses* (or all defined servers if *addresses* is omitted). The *mode* parameter specifies the new exposure mode as a string. Valid values are:

- 'antishake'
- 'auto'
- 'backlight'
- 'beach'
- 'fireworks'
- 'fixedfps'
- 'night'
- 'nightpreview'
- 'off'
- 'snow'
- 'sports'
- 'spotlight'
- 'verylong'

Note: When *mode* is set to 'off' the analog and digital gains reported by *status()* will become fixed. Any other mode causes them to vary according to the selected algorithm. Unfortunately, at

present, the camera firmware provides no means for forcing the gains to a particular value (in contrast to AWB and exposure speed).

awb(*mode*, *red=0.0*, *blue=0.0*, *addresses=None*)
Called to change the white balance on the servers at the specified *addresses* (or all defined servers if *addresses* is omitted). The *mode* parameter specifies the new white balance mode as a string. Valid values are:

- 'auto'
- 'cloudy'
- 'flash'
- 'fluorescent'
- 'horizon'
- 'incandescent'
- 'off'
- 'shade'
- 'sunlight'
- 'tungsten'

If the special value 'off' is given as the *mode*, the *red* and *blue* parameters specify the red and blue gains of the camera manually as floating point values between 0.0 and 8.0. Reasonable values for red and blue gains can be discovered easily by setting *mode* to 'auto', waiting a while to let the camera settle, then querying the current gain by calling *status()*. For example:

```
from time import sleep
from compoundpi.client import CompoundPiClient

with CompoundPiClient() as client:
    client.servers.network = '192.168.0.0/24'
    client.servers.find(10)
    # Pick an arbitrary camera to determine white balance gains and
    # set it auto white balance
    addr = client.servers[0]
    client.awb('auto', addresses=addr)
    # Wait a few seconds to let the camera measure the scene
    sleep(2)
    # Query the camera's gains and fix all cameras gains accordingly
    status = client.status(addresses=addr)[addr]
    client.awb('off', status.awb_red, status.awb_blue)
```

brightness(*value*, *addresses=None*)
Called to change the brightness level on the servers at the specified *addresses* (or all defined servers if *addresses* is omitted). The new level is specified an integer between 0 and 100.

capture(*count=1*, *video_port=False*, *quality=None*, *delay=None*, *addresses=None*)
Called to capture images on the servers at the specified *addresses* (or all defined servers if *addresses* is omitted). The optional *count* parameter is an integer value defining how many sequential images to capture, which defaults to 1. The optional *video_port* parameter defaults to False which indicates that the camera's slow, but high quality still port should be used for capture. If set to True, the faster, lower quality video port will be used instead. This is particularly useful with *count* greater than 1 for capturing high motion scenes.

The optional *delay* parameter defaults to None which indicates that all servers should capture images immediately upon receipt of the *CAPTURE* message. When using broadcast messages (when *addresses* is omitted) this typically results in near simultaneous captures, especially with fast, low latency networks like ethernet.

If *delay* is set to a small floating point value measured in seconds, it indicates that the servers should synchronize their captures to a timestamp (the client calculates the timestamp as *now* + *delay* seconds). This functionality assumes that the servers all have accurate clocks which are reasonably in sync with the client's clock; a typical configuration is to run an NTP server on the client machine, and an NTP client on each of the Compound Pi servers.

Note: Note that this method merely causes the servers to capture images. The captured images are stored in RAM on the servers for later retrieval with the *download()* method.

---

clear(*addresses=None*)
Called to clear captured files from the RAM of the servers at the specified *addresses* (or all defined servers if *addresses* is omitted). Currently the protocol for the *CLEAR* message is fairly crude: it simply clears all captured files on the server; there is no method for specifying a subset of files to wipe.

close()
Closes the client. This must be called once you are finished using the client to ensure that the background thread used for receiving downloaded data is shut down along with its listening socket. You can use the class as a context handler to ensure this happens easily:

```
import compoundpi.client

with compoundpi.client.CompoundPiClient() as client:
    # When this block terminates, close() will be called
    # implicitly
    client.servers.find(10)
```

contrast(*value*, *addresses=None*)
Called to change the contrast level on the servers at the specified *addresses* (or all defined servers if *addresses* is omitted). The new level is specified an integer between -100 and 100.

denoise(*value*, *addresses=None*)
Called to change whether the firmware's denoise algorithm is active on the servers at the specified *addresses* (or all defined servers if *addresses* is omitted). The *value* is a simple boolean, which defaults to `True`.

download(*address*, *index*, *output*)
Called to download the image with the specified *index* from the server at *address*, writing the content to the file-like object provided by the *output* parameter.

The *download()* method differs from all other client methods in that it targets a single server at a time (attempting to simultaneously download files from multiple servers would be extremely inefficient). The available image indices can be determined by calling the *list()* method beforehand. Note that downloading files from servers does *not* wipe the file from the server's RAM. Once all files have been successfully retrieved, you should use the *clear()* method to free up memory on the servers. For example:

```
import io
from compoundpi.client import CompoundPiClient

with CompoundPiClient() as client:
    client.servers.network = '192.168.0.0/24'
    # Capture an image on all servers
    client.capture()
    # Download all available files from all servers
    for addr, files in client.list().items():
        for f in files:
            print('Downloading image %d from %s (%d bytes)' % (
                f.index,
                addr,
                f.size,
```

```
                    ))
                with io.open('%s-%d.jpg' % (addr, f.index)) as f:
                    client.download(addr, f.index, f)
        # Wipe all files on all servers
        client.clear()
```

ev(*value*, *addresses=None*)
Called to change the exposure compensation (EV) level on the servers at the specified *addresses* (or all defined servers if *addresses* is omitted). The new level is specified an integer between -24 and 24 where each increment represents 1/6th of a stop.

exposure(*mode*, *speed=0*, *addresses=None*)
Called to change the exposure on the servers at the specified *addresses* (or all defined servers if *addresses* is omitted). The *mode* parameter specifies the new exposure mode as a string. Valid values are:

- `'auto'`
- `'off'`

The *speed* parameter specifies the exposure speed manually as a floating point value measured in milliseconds. Reasonable exposure speeds can be discovered easily by setting *mode* to `'auto'`, waiting a while to let the camera settle, then querying the current speed by calling *status()*. For example:

```
from time import sleep
from compoundpi.client import CompoundPiClient

with CompoundPiClient() as client:
    client.servers.network = '192.168.0.0/24'
    client.servers.find(10)
    # Pick an arbitrary camera to determine exposure speed and set it
    # to auto
    addr = client.servers[0]
    client.exposure('auto', addresses=addr)
    # Wait a few seconds to let the camera measure the scene
    sleep(2)
    # Query the camera's exposure speed and fix all cameras accordingly
    status = client.status(addresses=addr)[addr]
    client.exposure('off', speed=status.exposure_speed)
```

flip(*horizontal*, *vertical*, *addresses=None*)
Called to change the orientation of the servers at the specified *addresses* (or all defined servers if *addresses* is omitted). The *horizontal* and *vertical* parameters are boolean values indicating whether to flip the camera's output along the corresponding axis. The default for both parameters is `False`.

framerate(*rate*, *addresses=None*)
Called to change the camera framerate on the servers at the specified *addresses* (or all defined servers if *addresses* is omitted). The *rate* parameter is the new framerate specified as a numeric value (e.g. `int()`, `float()` or `Fraction`). For example:

```
from compoundpi.client import CompoundPiClient

with CompoundPiClient() as client:
    client.servers.network = '192.168.0.0/24'
    client.servers.find(10)
    client.framerate(24)
```

identify(*addresses=None*)
Called to cause the servers at the specified *addresses* to physically identify themselves (or all defined servers if *addresses* is omitted). Currently, the identification takes the form of the server blinking the camera's LED for 5 seconds.

iso(*value*, *addresses=None*)

Called to change the ISO setting on the servers at the specified *addresses* (or all defined servers if *addresses* is omitted). The *mode* parameter specifies the new ISO settings as an integer value. values are 0 (meaning auto), 100, 200, 320, 400, 500, 640, and 800.

list(*addresses=None*)
Called to list files available for download from the servers at the specified *addresses* (or all defined servers if *addresses* is omitted). The method returns a mapping of address to sequences of *CompoundPiFile* which provide the index, capture timestamp, and size of each image available on the server. For example, to enumerate the total size of all files stored on all servers:

```
from compoundpi.client import CompoundPiClient

with CompoundPiClient() as client:
    client.servers.network = '192.168.0.0/24'
    client.servers.find(10)
    client.capture()
    size = sum(
        f.size
        for addr, files in client.list().items()
        for f in files
        )
    print('%d bytes available for download' % size)
```

metering(*mode*, *addresses=None*)
Called to change the metering algorithm on the servers at the specified *addresses* (or all defined servers if *addresses* is omitted). The *mode* parameter specifies the new metering mode as a string. Valid values are:

- `'average'`
- `'backlit'`
- `'matrix'`
- `'spot'`

record(*length*, *format=u'h264'*, *quality=None*, *bitrate=None*, *intra_period=None*, *motion_output=False*, *delay=None*, *addresses=None*)
Called to record video on the servers at the specified *addresses* (or all defined servers if *addresses* is omitted). The *length* parameter specifies the time (in seconds) to record for. This may be a decimal value. The optional *format* parameter specifies the video codec to use. This defaults to `'h264'` but may also be set to `'mjpeg'`.

The optional *quality* parameter specifies the quality that the codec will attempt to maintain. This is an integer value between 1 and 40 for `h264` (lower values are better), or an integer value between 1 and 100 for `mjpeg` (higher values are better). The default provides "good" quality. The optional *bitrate* parameter specifies the limit of data that the codec is allowed to produce. The default is extremely high to ensure bitrate limiting never occurs by default.

The optional *intra_period* parameter is only valid with the `h264` format and specifies the number of frames in a GOP (group of pictures). As a GOP always starts with a keyframe (I-frame) this effectively dictates how regularly keyframes occurs in the output. The default is 30 frames.

The optional *motion_output* parameter is only valid with the `h264` format and specifies that you wish to capture motion vector estimation data as well as video data. This will be stored in a separate file on the Compound Pi server.

The optional *delay* parameter defaults to `None` which indicates that all servers should record video immediately upon receipt of the *CAPTURE* message. When using broadcast messages (when *addresses* is omitted) this typically results in near simultaneous recording, especially with fast, low latency networks like ethernet.

Note: Note that this method merely causes the servers to record video. The captured video is stored

in RAM on the servers for later retrieval with the *download()* method.

resolution(*width*, *height*, *addresses=None*)
Called to change the camera resolution on the servers at the specified *addresses* (or all defined servers if *addresses* is omitted). The *width* and *height* parameters are integers defining the new resolution. For example:

```
from compoundpi.client import CompoundPiClient

with CompoundPiClient() as client:
    client.servers.network = '192.168.0.0/24'
    client.servers.find(10)
    client.resolution(1280, 720)
```

saturation(*value*, *addresses=None*)
Called to change the saturation level on the servers at the specified *addresses* (or all defined servers if *addresses* is omitted). The new level is specified an integer between -100 and 100.

status(*addresses=None*)
Called to determine the status of servers. The *status()* method queries all servers at the specified *addresses* (or all defined servers if *addresses* is omitted) for their camera configurations. It returns a mapping of address to *CompoundPiStatus* named tuples. For example:

```
from compoundpi.client import CompoundPiClient

with CompoundPiClient() as client:
    client.servers.network = '192.168.0.0/24'
    client.servers.find(10)
    print('Configured resolutions:')
    for address, status in client.status().items():
        print('%s: %dx%d' % (
            address,
            status.resolution.width,
            status.resolution.height,
            ))
```

bind
Defines the port and interfaces the client will listen to for responses.

This attribute defaults to `('0.0.0.0', 5647)` meaning that the client defaults to listening on port 5647 on all available network interfaces for responses from Compound Pi servers (the special address `0.0.0.0` means "all available interfaces"). If you wish to change the port, or limit the interfaces the client listens to, assign a tuple of `(address, port)` to this attribute. For example:

```
from compoundpi.client import CompoundPiClient

with CompoundPiClient() as client:
    client.bind = ('192.168.0.1', 8000)
```

Querying this attribute will return a 2-tuple of the current address and port that the client is listening on.

Note: The port of the client's bound socket doesn't need to match the server's port. Both simply default to 5647 for the sake of simplicity.

servers
Stores the list of servers that the client controls.

See *CompoundPiServerList* for full documentation of the methods of the server list. For most purposes you can treat this as a normal Python list (e.g. *append()*, *remove()*, along with item access, length, etc). However, duplicate entries are not permitted, and there are a few extra methods like *find()* and *move()*.

This property is also writeable; setting it to a list of addresses will cause the server list to insert, remove, and move addresses as necessary to match the specified list. For example, this is a valid way to add a series of addresses to the list:

```
import compoundpi.client

with compoundpi.client.CompoundPiClient() as client:
    client.servers = [
        '192.168.0.%d' % i for i in range(1, 11)]
```

2.7.2 CompoundPiServerList

class compoundpi.client.**CompoundPiServerList**(*progress*)

Manages the list of servers under the control of the client.

The server list can be accessed via the *CompoundPiClient.servers* attribute. The list of defined servers can be manipulated with the familiar *append()*, *remove()*, and *extend()* methods, and individual entries can be replaced by assigning to them or deleted with del in the usual manner. The *find()* method can be used to discover available servers on the subnet via broadcast.

The list can be iterated over as usual, in reverse order with reversed(), and can be sorted with the *sort()* method just like a normal list.

Where the server list differs from a typical Python list is firstly that no duplicate addresses are permitted (in this manner, it is akin to a set). Secondly, while addresses can be added in string format, all addresses within the list will be converted to IPv4Address instances (which can be coerced back to strings for display purposes).

Furthermore, a *move()* method is provided to reposition existing addresses within the list. This is provided because adding new addresses to the list (via *append()*, *extend()*, or *find()* implicitly causes a *HELLO* message to be transmitted to the new servers to ensure they are alive and understand the correct version of the network protocol), so removing then re-inserting existing entries to move them is inefficient, whilst re-inserting then removing isn't permitted due to the prevention of duplicates.

You may also assign to the *CompoundPiClient.servers* attribute to re-order or completely redefine the list. Re-ordering in this case will be done efficiently.

> **Warning:** Upon construction, the assumes the local network is 192.168.0.0/16. Because this class utilizes UDP broadcast packets, it is crucial that the network configuration (including the network mask) is set correctly. If the default network is wrong (which is most likely the case), you must correct it before issuing any commands. This can be done by setting the *network* attribute.

The class assumes the servers are listening on UDP port 5647 by default. This can be altered via the *port* attribute.

append(*address*)

Called to explicitly add a server *address* to the client's list. This is equivalent to insertion at the end of the list:

```
from compoundpi.client import CompoundPiClient

with CompoundPiClient() as client:
    client.servers.network = '192.168.0.0/24'
    client.servers.append('192.168.0.2')
    assert len(client.servers) == 1
    assert '192.168.0.2' in client.servers
```

Attempting to add an address that is already present in the client's list will raise a CompoundPiRedefinedServer error.

extend(*addresses*)
Called to add multiple servers to the client's list. The *addresses* parameter must be an iterable of addresses to add.

find(*count=0*)
Called to discover servers on the client's network. The *find()* method broadcasts a *HELLO* message to the currently configured network. If called with no expected *count*, the method then waits for the network *timeout* (default 15 seconds) and adds all servers that replied to the broadcast to the client's list. If called with an expected *count* value, the method will terminate as soon as *count* servers have replied.

Note: If *count* servers don't reply, no exception will be raised. Therefore it is important to check the length of the list after calling *find()*.

For example:

```
from compoundpi.client import CompoundPiClient

with CompoundPiClient() as client:
    client.servers.network = '192.168.2.0/24'
    client.servers.find(10)
    assert len(client.servers) == 10
    print('Found 10 clients:')
    for addr in client.servers:
        print(str(addr))
```

This method or the *append()* method are usually the first methods called after construction and configuration of the client instance.

insert(*index*, *address*)
Called to explicitly add a server *address* to the client's list at the specified *index*. Before the server is added, the client will send a *HELLO* to verify that the server is alive. You can query the servers in the client's list by treating the list as an iterable:

```
from compoundpi.client import CompoundPiClient

with CompoundPiClient() as client:
    client.servers.network = '192.168.0.0/24'
    client.servers.insert(0, '192.168.0.2')
    assert len(client.servers) == 1
    assert '192.168.0.2' in client.servers
```

Attempting to add an address that is already present in the client's list will raise a `CompoundPiRedefinedServer` error.

move(*index*, *address*)
Called to move *address* (which must already be present within the server list) to *index*. Positioning is as for *insert()*; the specified *address* will be moved so that it occupies *index* and all later entries will be moved down.

remove(*address*)
Called to explicitly remove a server *address* from the client's list. Nothing is sent to a server that is removed from the list. If the server is still active on the client's network after removal it will continue to receive broadcast packets but the client will ignore any responses from the server.

> **Warning:** Please note that this may cause unexpected issues. For example, such a server (active but unknown to a client) may capture images in response to a broadcast *CAPTURE* message. For this reason it is recommended that you shut down any servers that you do not intend to communicate with. Future versions of the protocol may include explicit disconnection messages to mitigate this issue.

Attempting to remove an address that is not present in the client's list will raise a `ValueError`.

reverse()
Reverses the order of the servers in the list.

sort(*key=None*, *reverse=False*)
Sorts the servers in the list according to the specified *key* comparison function. If *reverse* is True, the order of the sort is reversed.

network
Defines the network that all servers belong to.

This attribute defaults to `192.168.0.0/16` meaning that the client assumes all servers belong to the network beginning with `192.168.` and accept broadcast packets with the address `192.168.255.255`. If this is incorrect (which is likely the case), assign the correct network configuration as a string (in CIDR or network/mask notation) to this attribute. A common configuration is:

```
from compoundpi.client import CompoundPiClient

with CompoundPiClient() as client:
    client.servers.network = '192.168.0.0/24'
```

Note that the network mask *must* be correct for broadcast packets to operate correctly. It is not enough for the network prefix alone to be correct.

Querying this attribute will return a `IPv4Network` object which can be converted to a string, or enumerated to discover all potential addresses within the defined network.

port
Defines the server port that the client will broadcast to.

This attribute defaults to 5647 meaning that the client will send broadcasts to Compound Pi servers which are assumed to be listening for messages on port 5647. If you have configured *cpid* differently, simply assign a different value to this attribute. For example:

```
from compoundpi.client import CompoundPiClient

with CompoundPiClient() as client:
    client.servers.port = 8080
```

Note: The port of the client's bound socket (see *CompoundPiClient.bind* doesn't need to match the server's port. Both simply default to 5647 for the sake of simplicity.

timeout
Defines the timeout for responses to commands.

This attribute specifies the length of time that the client will wait for all servers to complete a command and return a response. If all servers have not replied within the specified number of seconds a `CompoundPiTransactionFailed` error will be raised.

2.7.3 CompoundPiStatus

class compoundpi.client.**CompoundPiStatus**(*resolution*, *framerate*, *awb_mode*, ...)
This class is a namedtuple derivative used to store the status of a Compound Pi server. It is recommended you access the information stored by this class by attribute name rather than position (for example: `status.resolution` rather than `status[0]`).

resolution
Returns the current resolution of the camera as a *Resolution* tuple.

framerate
Returns the current framerate of the camera as a `Fraction`.

awb_mode
Returns the current white balance mode of the camera as a lower case string. See *CompoundPiClient.awb()* for valid values.

awb_red
Returns the current red gain of the camera's white balance as a floating point value. If *awb_mode* is `'off'` this is a fixed value. Otherwise, it is the current gain being used by the configured auto white balance mode.

awb_blue
Returns the current blue gain of the camera's white balance as a floating point value. If *awb_mode* is `'off'` this is a fixed value. Otherwise, it is the current gain being used by the configured auto white balance mode.

agc_mode
Returns the current auto-gain mode of the camera as a lower case string. See *CompoundPiClient.agc()* for valid values.

agc_analog
Returns the current analog gain applied by the camera. If *agc_mode* is `'off'` this is a fixed (but uneditable) value. Otherwise, it is a value which varies according to the selected AGC algorithm.

agc_digital
Returns the current digital gain used by the camera. If *agc_mode* is `'off'` this is a fixed (but uneditable) value. Otherwise, it is a value which varies according to the selected AGC algorithm.

exposure_mode
Returns the current exposure mode of the camera as a lower case string. See *CompoundPiClient.exposure()* for valid values.

exposure_speed
Returns the current exposure speed of the camera as a floating point value measured in milliseconds.

iso
Returns the camera's ISO setting as an integer value. This will be one of 0 (indicating automatic), 100, 200, 320, 400, 500, 640, or 800.

metering_mode
Returns the camera's metering mode as a lower case string. See *CompoundPiClient.metering()* for valid values.

brightness
Returns the camera's brightness level as an integer value between 0 and 100.

contrast
Returns the camera's contrast level as an integer value between -100 and 100.

saturation
Returns the camera's saturation level as an integer value between -100 and 100.

ev
Returns the camera's exposure compensation value as an integer value measured in 1/6ths of a stop. Hence, 24 indicates the camera's compensation is +4 stops, while -12 indicates -2 stops.

hflip
Returns a boolean value indicating whether the camera's orientation is horizontally flipped.

vflip
Returns a boolean value indicating whether the camera's orientation is vertically flipped.

denoise
Returns a boolean value indicating whether the camera's denoise algorithm is active when capturing.

timestamp
Returns a `datetime` instance representing the time at which the server received the *STATUS* message. Due to network latencies there is little point comparing this to the client's current timestamp.

However, if the *STATUS* message was broadcast to all servers, it can be useful to calculate the maximum difference in the server's timestamps to determine whether any servers have lost time sync.

files
Returns an integer number indicating the number of files currently stored in the server's memory.

2.7.4 CompoundPiFile

class compoundpi.client.**CompoundPiFile**(*filetype*, *image*, *timestamp*, *size*)
This class is a namedtuple derivative used to store information about an files stored in the memory of a Compound Pi server. It is recommended you access the information stored by this class by attribute name rather than position (for example: `f.size` rather than `f[3]`).

filetype
Specifies what sort of file this is. Can be one of `IMAGE`, `VIDEO`, or `MOTION`.

index
Specifies the index of the file on the server. This is the index that should be passed to *CompoundPiClient.download()* in order to retrieve this file.

timestamp
Specifies the timestamp on the server at which the file was captured as a `datetime` instance.

size
Specifies the size of the file as an integer number of bytes.

2.7.5 Resolution

class compoundpi.client.**Resolution**(*width*, *height*)
Represents an image resolution.

width
The width of the resolution as an integer value.

height
The height of the resolution as an integer value.

2.7.6 Examples

The following example demonstrates instantiating a client which attempts to find 10 Compound Pi servers on the 192.168.0.0/24 network. It configures all servers to capture images at 720p, captures a single image and then downloads the resulting images to the current directory, naming each image after the IP address of the server that captured it. Finally, the script ensures it clears all images from the servers:

```
import io
from compoundpi.client import CompoundPiClient

with CompoundPiClient() as client:
    client.servers.network = '192.168.0.0/24'
    client.servers.find(10)
    assert len(client.servers) == 10
    client.resolution(1280, 720)
    client.capture()
    try:
        for addr, files in client.list().items():
            for f in files:
                assert f.filetype == 'IMAGE'
                print('Downloading from %s' % addr)
                with io.open('%s.jpg' % addr, 'wb') as output:
                    client.download(addr, f.index, output)
```

```
    finally:
        client.clear()
```

The following example explicitly defines 5 servers, configures them with a variety of settings, then causes them to capture 5 images in rapid succession from their camera's video ports. The *delay* parameter of *CompoundPiClient.capture()* is used to synchronize the captures to a specific timestamp (it is assumed the servers clocks are synchronized). Finally, all images are downloaded into a series of in-memory streams (it is assumed the client has sufficient RAM to make this efficient):

```
import io
from compoundpi.client import CompoundPiClient

with CompoundPiClient() as client:
    client.servers.network = '192.168.0.0/24'
    print('Define servers 192.168.0.2-192.168.0.6')
    for i in range(2, 7):
        client.servers.append('192.168.0.%d' % i)
    assert len(client.servers) == 5
    print('Configuring servers')
    client.resolution(1280, 720)
    client.framerate(24)
    client.agc('auto')
    client.awb('off', 1.5, 1.3)
    client.iso(100)
    client.metering('spot')
    client.brightness(50)
    client.contrast(0)
    client.saturation(0)
    client.denoise(False)
    print('Capturing 5 images on all servers after 0.25 second delay')
    client.clear()
    client.capture(5, video_port=True, delay=0.25)
    for addr, status in client.status().items():
        assert status.files == 5
    print('Downloading captures')
    captures = {}
    try:
        for addr, files in client.list().items():
            for f in files:
                assert f.filetype == 'IMAGE'
                print('Downloading capture %d from %s' % (f.index, addr))
                stream = io.BytesIO()
                captures.setdefault(addr, []).append(stream)
                client.download(addr, f.index, stream)
                stream.seek(0)
    finally:
        client.clear()
```

The following example uses the *CompoundPiStatus.timestamp* field to determine whether the time on any of the discovered servers deviates from any other server by more than 0.1 seconds:

```
from compoundpi.client import CompoundPiClient

with CompoundPiClient() as client:
    client.servers.network = '192.168.0.0/24'
    client.servers.find(10)
    assert len(client.servers) == 10
    responses = client.status()
    min_time = min(status.timestamp for status in responses.values())
    for address, status in responses.items():
        if (status.timestamp - min_time).total_seconds() > 0.1:
            print(
                'Warning: time on %s deviates from minimum '
```

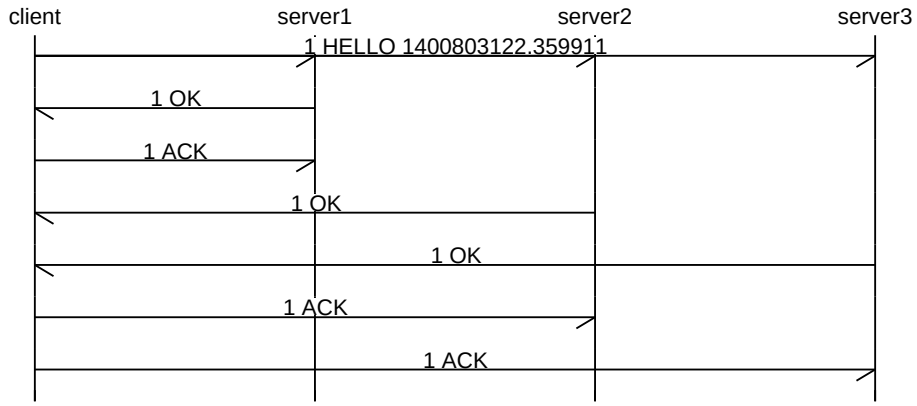
```
                'by >0.1 seconds' % address)
```

For more comprehensive examples, you may wish to browse the implementations of the *cpi* and cpigui applications as these are built using the *CompoundPiClient* class.

2.8 Network Protocol

Warning: As Compound Pi is a project in its infancy, the protocol version is currently the project's version and no attempt will be made to preserve backward (or forward) compatibility in the protocol until version 1.0 is released. In the current version, the client crudely compares the version in the *HELLO* response with its own version and rejects anything that doesn't match precisely.

The Compound Pi network protocol is UDP-based, utilizing broadcast or unicast packets for commands, and unicast packets for responses. File transfers (as initiated by the *download* command in the client) are TCP-based. The diagram below shows a typical conversation between a Compound Pi client and three servers involving a broadcast PING packet and the resulting responses:



All messages are encoded as ASCII text. Command messages consist of a non-zero positive integer sequence number followed by a single space, followed by the command in capital letters, optionally followed by comma-separated parameters for the command. The following are all valid examples of command messages:

```
1 HELLO 1400803122.359911

2 CLEAR

3 CAPTURE 1,0

4 STATUS

5 LIST

6 SEND 0,5647

7 FOO
```

In other words, the generic form of a command message is:

```
<sequence-number> <command> [parameter1],[parameter2],...
```

Response messages (from the servers to the client) consist of a non-zero positive integer sequence number (copied from the corresponding command), followed by a single space, followed by `OK` if the command's execution was successful, optionally followed by a new-line character (ASCII character 10), and any data the response is expected to include. For example:

```
1 OK
VERSION 0.4

2 OK

3 OK

4 OK
RESOLUTION 1280,720
FRAMERATE 30
AWB auto,1.5,1.3
AGC auto,2.3,1.0
EXPOSURE auto,28196
ISO 0
METERING average
BRIGHTNESS 50
CONTRAST 0
SATURATION 0
FLIP 0 0
EV 0
FLIP 0,0
DENOISE 1
TIMESTAMP 1400803173.991651
IMAGES 1

5 OK
IMAGE,0,1400803173.012543,8083879

6 OK
```

In the case of an error, the response message consists of a non-zero positive integer sequence number (copied from the corresponding command), followed by a single space, followed by `ERROR`, followed by a new-line character (ASCII character 10), followed by a description of the error that occurred:

```
7 ERROR
Unknown command FOO
```

In other words, the general form of a response message is:

```
<sequence-number> OK
<data>
```

Or, if an error occurred:

```
<sequence-number> ERROR
<error-description>
```

Sequence numbers start at 1 (0 is reserved), and are incremented on each command, except for protocol_ack and *HELLO*. The sequence number for a response indicates which command the response is associated with and likewise the sequence number for protocol_ack indicates the response that the protocol_ack terminates. The *HELLO* command, being the command that begins a session specifies a new starting sequence number for the server.

As UDP is an unreliable protocol, some mechanism is required to compensate for lost, unordered, or duplicated packets. All transmissions (commands and responses) are repeated with random delays. The sequence number associated with a client command permits servers to ignore repeated commands that they have already seen. Likewise, the sequence number of the server response permits clients to ignore repeated responses they have already seen.

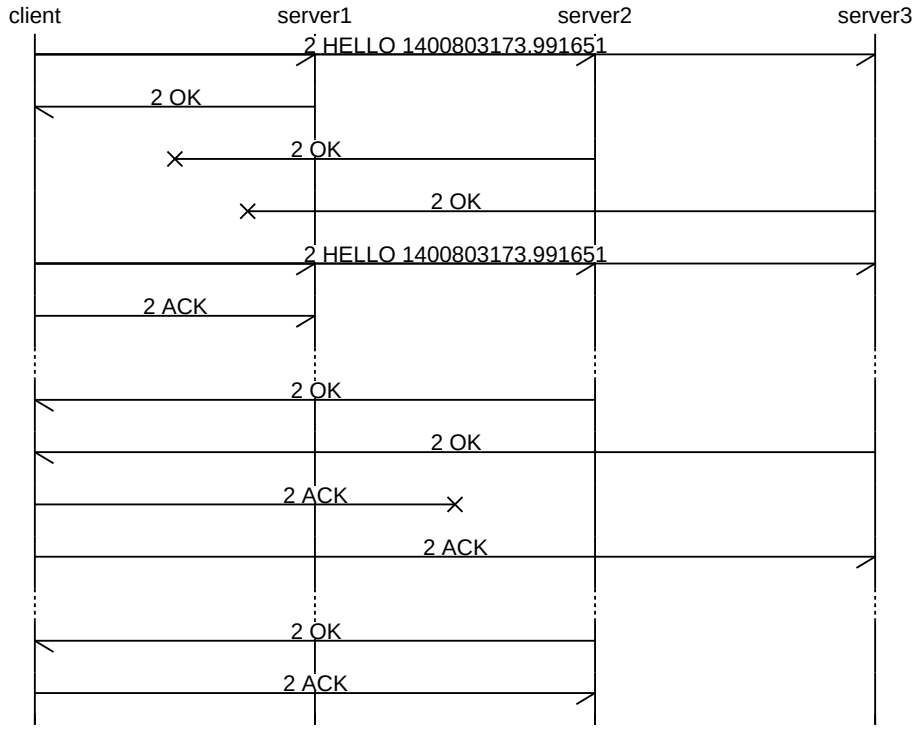
Commands are repeated by the client until it has received a response from the targetted server(s) (all located servers on the subnet in the case of broadcast messages), or until a timeout has elapsed (5 seconds by default).

Responses are repeated by a server until it receives an ACK from the client with a corresponding sequence number, or until a timeout has elapsed (5 seconds by default).

An exception to the above is the *HELLO* command. Because this command sets a new sequence number, servers cannot use the sequence number to detect repeated packets. Hence, the *HELLO* command includes the timestamp at the client issuing it as a command parameter. Servers must use this timestamp to detect stale or repeated instances of this messsage. The timestamp can be assumed to be incrementing (like a monotonic clock); in the current implementation it isn't but this doesn't matter much given how rarely this message is issued in a session.

2.8.1 Example

In the following example, the client broadcasts a *HELLO* command to three servers. The servers all respond with an OK response, but only the packet from server1 makes it back to the client. The server resends the HELLO command but this is ignored by the servers as they've seen the included timestamp before. The client responds to server1 with an protocol_ack. The other servers (after a random delay) now retry their OK responses and both get through this time. The client responds with an ACK for server3, but the ACK for server2 is lost. After another random delay, server2 once again retries its OK response, causing the client to send another ACK which succeeds this time:



The following sections document the various commands that the server understands and the expected responses.

2.8.2 AGC

Syntax: AGC *mode*

The *AGC* command changes the camera's auto-gain-control mode which is provided as a lower case string. If the string is `'off'` then the current sensor analog and digital gains will be fixed at their present values.

An OK response is expected with no data.

2.8.3 AWB

Syntax: AWB *mode,[red],[blue]*

The *AWB* command changes the camera's auto-white-balance mode which is provided as a lower case string. If the string is `'off'` then manual red and blue gains may additionally be specified as floating point values between 0.0 and 8.0.

An OK response is expected with no data.

2.8.4 BLINK

Syntax: BLINK

The *BLINK* command should cause the server to identify itself for the purpose of debugging. In this implementation, this is accomplished by blinking the camera's LED for 5 seconds.

An OK response is expected with no data.

2.8.5 BRIGHTNESS

Syntax: BRIGHTNESS *brightness*

The *BRIGHTNESS* command changes the camera's brightness. The new level is given as an integer number between 0 and 100 (default 50).

An OK response is expected with no data.

2.8.6 CAPTURE

Syntax: CAPTURE *[count],[use_video_port],[quality],[sync]*

The *CAPTURE* command should cause the server to capture one or more images from the camera. The parameters are as follows:

count Specifies the number of images to capture. If specified, this must be a non-zero positive integer number. If not specified, defaults to 1.

video-port Specifies which port to capture from. If unspecified, or 0, the still port should be used (resulting in the best quality capture, but may cause significant delay between multiple consecutive shots). If 1, the video port should be used.

quality Specifies the quality of the encoding. Valid values are 1 to 100 for `jpeg` encoding (larger is better).

sync Specifies the timestamp at which the capture should be taken. The timestamp's form is UNIX time: the number of seconds since the UNIX epoch specified as a dotted-decimal. The timestamp must be in the future, and it is important for the server's clock to be properly synchronized in order for this functionality to operate correctly. If unspecified, the capture should be taken immediately upon receipt of the command.

The image(s) taken in response to the command should be stored locally on the server until their retrieval is requested by the *SEND* command. The timestamp at which the image was taken must also be stored. Storage in this implementation is simply in RAM, but implementations are free to use any storage medium they see fit.

An OK response is expected with no data.

2.8.7 CLEAR

Syntax: CLEAR

The *CLEAR* command deletes all images from the server's local storage. As noted above in *CAPTURE*, implementations are free to use any storage medium, but the current implementation simply uses a list in RAM.

An OK response is expected with no data.

2.8.8 CONTRAST

Syntax: CONTRAST *contrast*

The *CONTRAST* command changes the camera's contrast. The new level is given as an integer number between -100 and 100 (default 0).

An OK response is expected with no data.

2.8.9 DENOISE

Syntax: DENOISE *denoise*

The *DENOISE* command changes whether the camera's software denoise algorithm is active (for both images and video). The new value is given as an integer which represents a boolean (0 being false, and anything else interpreted as true).

An OK response is expected with no data.

2.8.10 EV

Syntax: EV *ev*

The *SATURATION* command changes the camera's exposure compensation (EV). The new level is given as an integer number between -24 and 24 where increments of 6 represent one exposure stop.

An OK response is expected with no data.

2.8.11 EXPOSURE

Syntax: EXPOSURE *mode*,*speed*

The *EXPOSURE* command changes the camera's exposure mode, speed, and compensation value. The mode is provided as a lower case string. If the string is `'off'`, the speed may additionally be specified as a floating point number measured in milliseconds.

An OK response is expected with no data.

2.8.12 FLIP

Syntax: FLIP *horizontal*,*vertical*

The *FLIP* command changes the camera's orientation. The horizontal and vertical parameters must be integer numbers which will be interpreted as booleans (0 being false, anything else true).

An OK response is expected with no data.

2.8.13 FRAMERATE

Syntax: FRAMERATE *rate*

The *FRAMERATE* command changes the camera's configuration to use the specified framerate which is given either as an integer number between 1 and 90 or as a fraction consisting of an integer numerator and denominator separated by a forward-slash.

An OK response is expected with no data.

2.8.14 HELLO

Syntax: HELLO *timestamp*

The *HELLO* command is sent by the client's *find* command in order to locate Compound Pi servers. The server must send the following string in the data portion of the OK response indicating the version of the protocol that the server understands:

```
VERSION 0.4
```

The server must use the sequence number of the command as the new starting sequence number (i.e. HELLO resets the sequence number on the server). For this reason, the sequence number cannot be used to detect repeated HELLO commands. Instead the timestamp parameter should be used for this purpose: the timestamp can be assumed to be incrementing hence HELLO commands from a particular host with a timestamp less than or equal to one already seen can be ignored.

2.8.15 ISO

Syntax: ISO *iso*

The *ISO* command changes the camera's emulated ISO level. The new level is provided as an integer number where 0 indicates automatic ISO level.

An OK response is expected with no data.

2.8.16 LIST

Syntax: LIST

The *LIST* command causes the server to respond with a new-line separated list detailing all locally stored files. Each line in the data portion of the response has the following format:

```
<filetype>,<number>,<timestamp>,<size>
```

For example, if four images and one video are stored on the server the data portion of the OK response may look like this:

```
IMAGE,0,1398618927.307944,8083879
IMAGE,1,1398619000.53127,7960423
IMAGE,2,1398619013.658935,7996156
IMAGE,3,1398619014.122921,8061197
VIDEO,4,1398619014.314919,28053651
```

The filetype will be `IMAGE`, `VIDEO`, or `MOTION` depending on the type of data contained within.

The `number` portion of the line is a zero-based integer index for the image which can be used with the *SEND* command to retrieve the image data. The `timestamp` portion is in UNIX-time format: a dotted-decimal value of the number of seconds since the UNIX epoch. Finally, the `size` portion is an integer number indicating the number of bytes in the image.

2.8.17 METERING

Syntax: METERING *mode*

The *METERING* command changes the camera's light metering mode. The new mode is provided as a lower case string.

An OK response is expected with no data.

2.8.18 RECORD

Syntax: RECORD *length*,*[format]*,*[quality]*,*[bitrate]*,*[intra_period]*,*[motion_output]*,*[sync]*

The *RECORD* command should cause the server to record a video for *length* seconds from the camera. The parameters are as follows:

length Specifies the length of time to record for as a non-zero floating point number.

format Specifies the encoding to use. Valid values are `mjpeg` and `h264`.

quality Specifies the quality of the encoding. If unspecified or zero, a suitable default will be selected for the specified encoding. Valid values are 1 to 40 for `h264` encoding (smaller is better), and 1 to 100 for `mjpeg` encoding (larger is better).

bitrate Specifies the bitrate limit for the video encoder. Defaults to 17000000 if unspecified.

sync Specifies the timestamp at which the recording should begin. The timestamp's form is UNIX time: the number of seconds since the UNIX epoch specified as a dotted-decimal. The timestamp must be in the future, and it is important for the server's clock to be properly synchronized in order for this functionality to operate correctly. If unspecified, the recording should begin immediately upon receipt of the command.

intra-period Only valid if format is `h264`. Specifies the number of frames in a GOP (group of pictures), the first of which must be a keyframe (I-frame). Defaults to 30 if unspecified.

motion-output Only valid if format is `h264`. If unspecified or 0, only video data is output. If 1, motion estimation vector data is also recorded as a separate file with an equivalent timestamp to the corresponding video data.

The video recorded in response to the command should be stored locally on the server until its retrieval is requested by the *SEND* command. The timestamp at which the recording was started must be stored. Storage in this implementation is simply in RAM, but implementations are free to use any storage medium they see fit.

An OK response is expected with no data.

2.8.19 RESOLUTION

Syntax: RESOLUTION *width*,*height*

The *RESOLUTION* command changes the camera's configuration to use the specified capture resolution which is two integer numbers giving the width and height of the new resolution.

An OK response is expected with no data.

2.8.20 SATURATION

Syntax: SATURATION *saturation*

The *SATURATION* command changes the camera's saturation. The new level is given as an integer number between -100 and 100 (default 0).

An OK response is expected with no data.

2.8.21 SEND

Syntax: SEND *file_num*,*port*

The *SEND* command causes the specified file to be sent from the server to the client. The parameters are as follows:

index Specifies the zero-based index of the file that the client wants the server to send. This must match one of the indexes output by the *LIST* command.

port Specifies the TCP port on the client that the server should connect to in order to transmit the data. This is given as an integer number (never a service name).

Assuming *index* refers to a valid image file, the server must connect to the specified TCP port on the client, send the bytes of the file, and finally close the connection. The server must also send an OK response with no data.

2.8.22 STATUS

Syntax: STATUS

The *STATUS* command causes the server to send the client information about its current configuration. Specifically, the response must contain the following lines in its data portion, in the order given below:

```
RESOLUTION <width>,<height>
FRAMERATE <rate>
AWB <awb_mode>,<awb_red>,<awb_blue>
AGC <agc_mode>,<agc_analog>,<agc_digital>
EXPOSURE <exp_mode>,<exp_speed>
ISO <iso>
METERING <metering_mode>
BRIGHTNESS <brightness>
CONTRAST <contrast>
SATURATION <saturation>
EV <ev>
FLIP <hflip>,<vflip>
DENOISE <denoise>
TIMESTAMP <time>
IMAGES <images>
```

Where:

<width> <height> Gives the camera's currently configured capture resolution

<rate> Gives the camera's currently configured framerate as an integer number or fractional value (num/denom)

<awb_mode> Gives the camera's current auto-white-balance mode as a lower case string

<awb_red> Gives the camera's red-gain as an integer number or fractional value

<awb_blue> Gives the camera's blue-gain as an integer number or fractional value

<agc_mode> Gives the camera's current auto-gain-control mode as a lower case string

<agc_analog> Gives the camera's current analog gain as a floating point value

<agc_digital> Gives the camera's current digital gain as a floating point value

<exp_mode> Gives the camera's current exposure mode as a lower case string

<exp_speed> Gives the camera's current exposure speed as a floating point number measured in milliseconds.

<iso> Gives the camera's current ISO setting as an integer number between 0 and 1600 (where 0 indicates automatic)

<metering_mode> Gives the camera's current light metering mode as a lower case string

<brightness> Gives the camera's current brightness setting as an integer value between 0 and 100 (50 is the default)

<contrast> Gives the camera's current contrast setting as an integer between -100 and 100 (0 is the default)

<saturation> Gives the camera's current saturation setting as an integer between -100 and 100 (0 is the default)

<ev> Gives the camera's current exposure compensation value as an integer number between -24 and 24 (each increment represents 1/6th of a stop)

<hflip>* and *<vflip> Gives the camera's orientation as 1 or 0 (indicating the flip is or is not active respectively)

<denoise> Gives the camera's software denoise status as 1 or 0 (indicating denoise is active or not respectively)

<time> Gives the timestamp at which the *STATUS* command was received in UNIX time format (a dotted-decimal number of seconds since the UNIX epoch).

<images> Gives the number of images currently stored locally by the server.

For example, the data portion of the OK response may look like the following:

```
RESOLUTION 1280 720
FRAMERATE 30
AWB auto 321/256 3/2
AGC auto 8.0 1.5
EXPOSURE auto 33.158
ISO 0
METERING average
BRIGHTNESS 50
CONTRAST 0
SATURATION 0
EV 0
FLIP 0 0
DENOISE 1
TIMESTAMP 1400803173.991651
IMAGES 1
```

2.9 Change log

2.9.1 Release 0.4 (2015-08-24)

Major enhancements in this release:

- Fixed bug where restarting client quickly after quit would fail (#21)
- Added an officially documented batch interface (#22)
- Added ability to control denoise algorithm on servers (#23)
- Added video support to the protocol (accessible from command line and batch client, but not GUI) (#24)
- Added ability to copy settings from one server to all others (#25)
- Added ability to order servers; supported in all clients but only really useful in the batch client currently (#26)
- Added ability to configure quality of captures (#29)

2.9.2 Release 0.3 (2014-05-23)

Several major enhancements in this release:

- A GUI client (cpigui) is now included. This is currently undocumented, but should be pretty intuitive to anyone familiar with the command line interface (#3)
- Both clients and the server now support many more camera settings including white-balance, exposure, ISO, shutter speed, etc (#12)
- All UDP messages (client and server) are now retried to ensure reliability, particularly during multiple unicast messages (#13)

2.9.3 Release 0.2 (2014-04-27)

Several improvements in this release:

- The network protocol has been changed to enhance its reliability when dealing with lots of Pis on unreliable networks (like Wifi)
- The status command has been enhanced to warn of configuration discrepancies.

- Lots more work on the docs

2.9.4 Release 0.1 (2014-04-15)

Initial release

2.10 License

CHAPTER 3

Indices and Tables

- genindex
- search

C

Symbols

A

B

C

D

E

F

H

I

L

M

N

P

R

S

T

V

W