lightsd Documentation

Release 1.1.2

Louis Opter

Nov 06, 2017

Contents

Warning: The docs have moved to: https://docs.lightsd.io/.

Welcome! This is the documentation for lightsd: a small program that runs in the background, discovers smart bulbs[1] on your local network and let you control them easily.

lightsd exposes a JSON-RPC interface over TCP IPv4/IPv6[2] and Unix sockets. The same interface can be exposed over a named pipe: in that case responses can't be read back from lightsd but this is useful to control your lights from very basic shell scripts.

lightsd works out of the box on Mac OS X and Arch Linux but is very easy to build thanks to its very limited requirements. Check-out the installation instructions:

[1] Currently only LIFX WiFi smart bulbs are supported.
[2] And not over HTTP like most JSON-RPC implementations.

CHAPTER 1

Installation instructions

1.1 Installation (for Mac OS X)

Make sure you have brew installed and updated: http://brew.sh/.

```
brew install lopter/lightsd/lightsd
```

Or,

```
brew tap lopter/lightsd
brew install lightsd
```

Make sure you execute the `ln -sfv` command displayed at the end of the installation:

```
ln -sfv /usr/local/opt/lightsd/*.plist ~/Library/LaunchAgents
```

Please, also install Python 3 and ipython if you want to follow the examples in the next section:

```
brew install python3
pip3 install ipython
```

Read on *First steps* to see how to use lightsd.

1.2 Installation (for Arch Linux)

Make sure you have Yaourt installed: https://archlinux.fr/yaourt-en (wiki page).

```
yaourt -Sya lightsd
```

Make sure to follow the post-installation instructions: replace `$USER` with the user you usually use.

Please also install ipython if you want to follow the examples in the next section:

```
yaourt -Sya ipython
```

Read on *First steps* to see how to use lightsd.

1.3 Installation (OpenWRT trunk)

If you're running OpenWRT trunk then, from your build root, just add lightsd's feed:

```
cat >>feeds.conf`[ -f feeds.conf ] || echo .default` <<EOF
src-git lightsd https://github.com/lopter/openwrt-lightsd.git
EOF
./scripts/feeds update -a
```

Install lightsd:

```
./scripts/feeds install lightsd
```

Run your usual `make menuconfig`, `make` firmware flash flow, lightsd should be running at startup.

Read on *First steps* to see how to use lightsd.

1.4 Build instructions for Debian based systems (Ubuntu/Raspbian)

Note: Those instructions have been tested on Debian Wheezy & Jessie.

Install the following packages:

```
apt-get install build-essential cmake libevent-dev git ca-certificates ipython3␣
↪fakeroot wget devscripts debhelper
```

Download and extract lightsd:

```
wget -O lightsd_1.1.2.orig.tar.gz https://github.com/lopter/lightsd/archive/1.
↪1.2.tar.gz
tar -xzf lightsd_1.1.2.orig.tar.gz
cd lightsd-1.1.2
wget -O - https://github.com/lopter/lightsd/releases/download/1.1.2/dpkg-1.1.
↪2.tar.gz | tar -xzf -
```

Build the package:

```
debuild -uc -us
```

Install the package:

Note: You will need to run this command as root with sudo(8) or be logged in as root already.

```
dpkg -i ../lightsd_1.1.2-1_`dpkg --print-architecture`.deb
```

Still as root, run the command the package asks you to run:

Note: If you are *not using sudo* on your system replace `$USER` by your regular non-root username:

```
gpasswd -a $USER lightsd
```

Log out and back in as `$USER` for the change to take effect.

Read on *First steps* to see how to use lightsd.

1.5 Build instructions (for other systems)

lightsd should work on any slightly POSIX system (i.e: not Windows), make sure you have the following requirements installed:

- libevent 2.0.19 (released May 2012);
- CMake 2.8.9 (released August 2012).

lightsd is developed and tested from Arch Linux, Debian, OpenBSD and Mac OS X; both for 32/64 bits and little/big endian architectures.

Please also install ipython with Python 3 if you want to follow the examples in the next section.

From a terminal prompt, clone the repository and move to the root of it:

```
git clone https://github.com/lopter/lightsd.git
cd lightsd
```

From the root of the repository:

```
mkdir build && cd build
cmake -DCMAKE_BUILD_TYPE=RELEASE ..
make -j5 lightsd
```

Read on *First steps* to see how to use lightsd.

CHAPTER 2

First steps

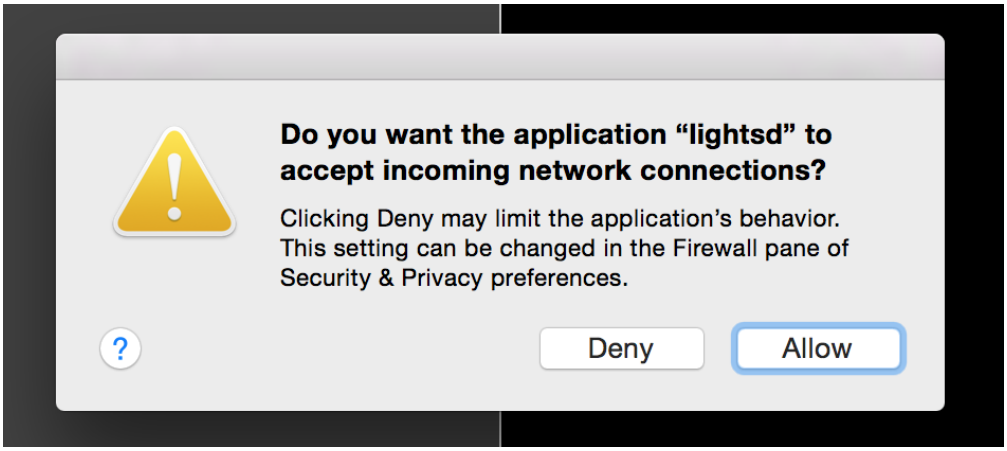
Those instructions assume that you have followed the *installation instructions*.

2.1 Starting and stopping lightsd

lightsd listens for UDP traffic from the bulbs on `0.0.0.0` port `udp/56700`.

2.1.1 Mac OS X

Note: This warning will be displayed the first time you start lightsd after an install or upgrade:



Click Allow, lightsd uses the network to communicate with your bulbs.

Start lightsd with:

```
launchctl load ~/Library/LaunchAgents/homebrew.mxcl.lightsd.plist
```

Stop lightsd with:

```
launchctl unload ~/Library/LaunchAgents/homebrew.mxcl.lightsd.plist
```

Check how lightsd is running with:

```
ps aux | grep lightsd
```

Read the logs with:

```
tail -F `brew --prefix`/var/log/lightsd.log
```

Try to *toggle your lights* and read on some of the examples bundled with lightsd.

2.1.2 Linux (systemd)

Start lightsd with:

```
systemctl start lightsd
```

Stop lightsd with:

```
systemctl stop lightsd
```

Enable lightsd at boot:

```
systemctl enable lightsd
```

Check how lightsd is running with:

```
ps aux | grep lightsd
```

Read the logs with:

```
journalctl -x -f _SYSTEMD_UNIT=lightsd.service
```

Try to *toggle your lights* and read on some of the examples bundled with lightsd.

2.1.3 Linux (System V style)

Start lightsd with:

```
/etc/init.d/lightsd start
```

Stop lightsd with:

```
/etc/init.d/lightsd stop
```

Check how lightsd is running with:

```
ps aux | grep lightsd
```

The logs will be logged to syslogd(8).

2.1.4 OpenWRT (procd)

Start lightsd with:

```
/etc/init.d/lightsd start
```

Stop lightsd with:

```
/etc/init.d/lightsd stop
```

Enable lightsd at boot:

```
/etc/init.d/lightsd enable
```

Check how lightsd is running with:

```
pgrep -l lightsd
```

Read the logs with:

```
logread -e lightsd -f
```

Try to *toggle your lights* and read on some of the examples bundled with lightsd.

2.1.5 Manually (other systems)

Assuming you've just built *lightsd from the sources*, lightsd will be in the `core` directory[1].

The examples are communicating with lightsd through a pipe or an Unix socket, start lightsd with them:

```
core/lightsd -c pipe -s socket
```

From another terminal, check how lightsd is running with:

```
ps aux | grep lightsd
```

You can stop lightsd with ^C (ctrl+c).

Checkout the *examples*.

2.1.6 Command line options

```
Usage: lightsd ...

  [-l,--listen addr:port [+]]            Listen for JSON-RPC commands over TCP at
                                         this address (can be repeated).
  [-c,--command-pipe /command/fifo [+]]  Open an unidirectional JSON-RPC
                                         command pipe at this location (can be
                                         repeated).
  [-s,--socket /unix/socket [+]]         Open an Unix socket at this location
                                         (can be repeated).
  [-d,--daemonize]                       Fork in the background.
  [-p,--pidfile /path/to/pid.file]       Write lightsd's pid in the given file.
  [-u,--user user]                       Drop privileges to this user (and the
```

[1] `build/core` if you start from the root of the repository.

```
                                        group of this user if -g is missing).
  [-g,--group group]                    Drop privileges to this group (-g requires
                                        the -u option to be used).
  [-S,--syslog]                         Divert logging from the console to syslog.
  [-F,--syslog-facility]                Facility to use with syslog (defaults to
                                        daemon, other possible values are user and
                                        local0-7, see syslog(3)).
  [-I,--syslog-ident]                   Identifier to use with syslog (defaults to
                                        lightsd).
  [-t,--no-timestamps]                  Disable timestamps in logs.
  [-h,--help]                           Display this.
  [-V,--version]                        Display version and build information.
  [-v,--verbosity debug|info|warning|error]

or,

  --prefix                              Display the install prefix for lightsd.

or,

  --rundir                              Display the runtime directory for lightsd.
```

2.2 Toggle your lights

```
`lightsd --prefix`/share/lightsd/examples/toggle
```

Or, from the root of the repository:

```
examples/toggle
```

Here is the source code of this example, it uses a small client —lightsc.sh— the next section covers it:

2.3 Using lightsc.sh

lightsc.sh is a small shell script that wraps a few things around lightsd' command pipe. Once you've sourced it with:

```
. `lightsd --prefix`/share/lightsd/lightsc.sh
```

Or, from the root of the repository:

```
. share/lightsc.sh
```

You can use the following variable and functions to send commands to your bulbs from your current shell or shell script:

LIGHTSD_COMMAND_PIPE
: By default lightsc will use `$(lightsd --rundir)/pipe` but you can set that to your own value.

lightsc method [arguments...]
: Call the given *method* with the given arguments. lightsc display the generated JSON that was sent.

lightsc_get_pipe
: Equivalent to `${LIGHTSD_COMMAND_PIPE:-$(lightsd --rundir)/pipe}` but also check if lightsd is running.

lightsc_make_request method [arguments...]
Like lightsc but display the generated json instead of sending it out to lightsd: with this and lightsc_get_pipe you can do batch requests:

Note: Keep in mind that arguments must be JSON, you will have to enclose tags and labels into double quotes `'"likethis"'`. The command pipe is write-only: you cannot read any result back.

Examples:

Build a batch request manually:

```
tee $(lightsc_get_pipe) <<EOF
[
    $(lightsc_make_request power_on '"#tag"'),
    $(lightsc_make_request set_light_from_hsbk '"#othertag"' 37.469443 1.0 0.05 3500␣
↪600),
    $(lightsc_make_request set_light_from_hsbk '["bulb","otherbulb"]' 47.469443 0.2 0.
↪05 3500 600)
]
EOF
```

2.4 Using lightsc.py

lightsc.py is a minimalistic Python client for lightsd, if you run it as a program it will open a python shell from which you can directly manipulate your bulbs. Start lightsc.py with:

```
`lightsd --prefix`/share/lightsd/examples/lightsc.py
```

Or, from the root of the repository:

```
examples/lightsc.py
```

From there, a `c` variable has been initialized for you: this small object lets you directly execute commands on your bulb:

For example toggle your lights again:

```
c.power_toggle("*")
```

Fetch the state of all your bulbs:

```
bulbs = {b["label"]: b for b in c.get_light_state("*")["result"]}
```

lightsc.py also accepts an url which lets you connect to anything running lightsd, e.g:

```
lightsc.py -u tcp://localhost:1234
```

Or, for an Unix socket:

```
lightsc.py -u unix:///path/to/lightsd/socket
```

Check out *lightsd's API* to see everything you can do!

CHAPTER 3

The lights daemon protocol

The lightsd protocol is implemented on top of JSON-RPC 2.0. This section covers the available methods and how to target bulbs.

Since lightsd implements JSON-RPC without any kind of framing like it usually is the case (using HTTP), this section also explains how to implement your own lightsd client in *Writing a client for lightsd*.

3.1 Targeting bulbs

Commands that manipulate bulbs will take a *target* argument to define on which bulb(s) the operation should apply. The target argument is either a string (identifying a target as explained in the following table), or an array of strings (targets).

`*`	targets all bulbs
`#TagName`	targets bulbs tagged with *TagName*
`124f31a5`	directly target the bulb with the given id (that's the bulb mac address, see below)
`label`	directly target the bulb with the given Label
`[#TagName, 123f31a5]`	composite target (JSON array)

The mac address (id) of each bulb can be found with *get_light_state* under the `_lifx` map, e.g:

```
"_lifx": {
    "addr": "d0:73:d5:02:e5:30",
    "gateway": {
        [...]
```

This bulb has id `d073d502e530`.

Note: The maximum supported length for labels and tag names by LIFX bulbs is 32. Anything beyond that will be ignored.

3.2 Available methods

power_off (*target*)
Power off the given bulb(s).

power_on (*target*)
Power on the given bulb(s).

power_toggle (*target*)
Power on (if they are off) or power off (if they are on) the given bulb(s).

set_light_from_hsbk (*target*, *h*, *s*, *b*, *k*, *t*)

Parameters

- **h** (*float*) – Hue from 0 to 360.
- **s** (*float*) – Saturation from 0 to 1.
- **b** (*float*) – Brightness from 0 to 1.
- **k** (*int*) – Temperature in Kelvin from 2500 to 9000.
- **t** (*int*) – Transition duration to this color in ms.

set_waveform (*target*, *waveform*, *h*, *s*, *b*, *k*, *period*, *cycles*, *skew_ratio*, *transient*)

Parameters

- **waveform** (*string*) – One of SAW, SINE, HALF_SINE, TRIANGLE, SQUARE.
- **h** (*float*) – Hue from 0 to 360.
- **s** (*float*) – Saturation from 0 to 1.
- **b** (*float*) – Brightness from 0 to 1.
- **k** (*int*) – Temperature in Kelvin from 2500 to 9000.
- **period** (*int*) – milliseconds per cycle.
- **cycles** (*int*) – number of cycles.
- **skew_ratio** (*float*) – from 0 to 1.
- **transient** (*bool*) – if true the target will keep the color it has at the end of the waveform, otherwise it will revert back to its original state.

The meaning of the skew_ratio argument depends on the type of waveform:

SAW	Should be 0.5.
SINE	Defines the peak point of the function, 0.5 gives you a sine and 1 or 0 will give you cosine. Ignored by firmware 1.1.
HALF_SINE	Should be 0.5.
TRIANGLE	Defines the peak point of the function like SINE. Ignored by firmware 1.1.
SQUARE	Ratio of a cycle the targets are set to the given color.

get_light_state (*target*)
Return a list of dictionnaries, each dict representing the state of one targeted bulb, the list is not in any specific order. Each dict has the following fields:

- hsbk: tuple (h, s, b, k) see function:*set_light_from_hsbk*;
- label: bulb label (utf-8 encoded string);
- power: boolean, true when the bulb is powered on false otherwise;

- tags: list of tags applied to the bulb (utf-8 encoded strings).

set_label (*target*, *label*)
Label the target bulb(s) with the given label.

Note: Use *tag()* instead set_label to give a common name to multiple bulbs.

---

tag (*target*, *label*)
Tag (group) the given target bulb(s) with the given label (group name), then label can be used as a target by prefixing it with #.

To add a device to an existing "group" simply do:

```
tag(["#myexistingtag", "bulbtoadd"], "myexistingtag")
```

Note: Notice how # is prepended to the tag label depending on whether it's used as a target or a regular argument.

---

untag (*target*, *label*)
Remove the given tag from the given target bulb(s). To completely delete a tag (group), simple do:

```
untag("#myexistingtag", "myexistingtag")
```

3.3 Writing a client for lightsd

lightsd does JSON-RPC directly over TCP, requests and responses aren't framed in any way like it is usually done by using HTTP.

This means that you will very likely need to write a JSON-RPC client specifically for lightsd. You're actually encouraged to do that as lightsd will probably augment JSON-RPC via lightsd specific JSON-RPC extensions in the future.

3.3.1 JSON-RPC over TCP

JSON-RPC works in a request/response fashion: the socket (network connection) is never used in a full-duplex fashion (data never flows in both direction at the same time):

1. Write (send) a request on the socket;
2. Read (receive) the response on the socket;
3. Repeat.

Writing the request is easy: do successive write (send) calls until you have successfully sent the whole request. The next step (reading/receiving) is a bit more complex. And that said, if the response isn't useful to you, you can ask lightsd to omit it by turning your request into a notification: if you remove the JSON-RPC id, then you can just send your requests (now notifications) on the socket in a fire and forget fashion.

Otherwise to successfully read and decode JSON-RPC over TCP you will need to implement your own read loop, the algorithm follows. It focuses on the low-level details, adapt it for the language and platform you are using:

1. Prepare an empty buffer that you can grow, we will accumulate received data in it;

2. Start an infinite loop and start a read (receive) for a chunk of data (e.g: 4KiB), accumulate the received data in the previous buffer, then try to interpret the data as JSON:
   - if valid JSON can be decoded then break out of the loop;
   - else data is missing and continue the loop;
3. Decode the JSON data.

Here is a complete Python 3 request/response example:

```
import json
import socket
import uuid

READ_SIZE = 4096
ENCODING = "utf-8"

# Connect to lightsd, here using an Unix socket. The rest of the example is
# valid for TCP sockets too. Replace /run/lightsd/socket by the output of:
# echo $(lightsd --rundir)/socket
lightsd_socket = socket.socket(socket.AF_UNIX)
lightsd_socket.connect("/run/lightsd/socket")
lightsd_socket.settimeout(2)  # seconds

# Prepare the request:
request = json.dumps({
    "method": "get_light_state",
    "params": ["*"],
    "jsonrpc": "2.0",
    "id": str(uuid.uuid4()),
}).encode(ENCODING, "surrogateescape")

# Send it:
lightsd_socket.sendall(request)

# Prepare an empty buffer to accumulate the received data:
response = bytearray()
while True:
    # Read a chunk of data, and accumulate it in the response buffer:
    response += lightsd_socket.recv(READ_SIZE)
    try:
        # Try to load the received the data, we ignore encoding errors
        # since we only wanna know if the received data is complete.
        json.loads(response.decode(ENCODING, "ignore"))
        break  # Decoding was successful, we have received everything.
    except Exception:
        continue  # Decoding failed, data must be missing.

response = response.decode(ENCODING, "surrogateescape")
print(json.loads(response))
```

3.3.2 Notes

- Use an incremental JSON parser if you have one handy: for responses multiple times the size of your receive window it will let you avoid decoding the whole response at each iteration of the read loop;
- lightsd supports batch JSON-RPC requests, use them!

CHAPTER 4

Known issues

All LIFX bulbs –except for the original model released with their kickstarter campaign (called the "Original 1000")– are suffering from a critical firmware bug[1]. Under some specific and unknown Wi-Fi conditions LIFX bulbs will crash, forcing you to turn them off and then back on using a physical light switch.

Crashed LIFX bulbs will show up as unavailable[2] in the LIFX mobile application and they will be missing in lightsd's `get_light_state` result list.

The only known workaround at this time is to try different Wi-Fi channel, you might find one that doesn't trigger the bug for your bulbs and radio-frequency environment (LIFX bulbs are running in the very busy 2.4GHz band).

Power ON/OFF are the only commands with auto-retry, i.e: lightsd will keep sending the command to the bulb until its state changes. This is not implemented (yet) for `set_light_from_hsbk`, `set_waveform`, `set_label`, `tag` and `untag`.

In general, crappy Wi-Fi network with latency, jitter or packet loss are gonna be challenging until lightsd has an auto-retry mechanism, there is also room for optimizations in how lightsd communicates with the bulbs.

[1] More accurately a firmware bug within the Qalcomm Atheros chip upon which all LIFX products released since 2015 are built.
[2] Kinda like this emoji: .

CHAPTER 5

Developers & companies

lightsd's development takes place on github: https://github.com/lopter/lightsd.

Check-out the contribution guide!

If you wanna get a feel of what I'm working on you can watch my patch queue: https://github.com/lopter/lightsd-mq. You will need to install Mercurial and hg-git to apply it over the main repository. Some of the patches are just snippets I put aside; the series file will tell you which ones.

Feel free to reach out via email or irc (kalessin on freenode, insist if I don't reply). As the project name implies, I'm fairly interested in other smart bulbs. If you are a company trying to use lightsd feel free to reach me out as well. lightsd is free software under the GPLv3 but has been designed in a way that make it usable in closed source environments.

I'm not looking for a new job, thanks.

CHAPTER 6

Packaging lightsd

lightsd has already been packaged for:

- Mac OS X's Homebrew: https://github.com/lopter/homebrew-lightsd;
- Arch Linux (AUR): https://aur.archlinux.org/packages/lightsd/;
- Debian/Ubuntu: https://github.com/lopter/dpkg-lightsd;
- OpenWRT: https://github.com/lopter/openwrt-lightsd.

Here is what you need to know to build lightsd in order to be distributed:

- make sure CMake's build type is set to `RELEASE`;
- lightsd needs a runtime directory, this must be configured via CMake using `LGTD_RUNTIME_DIRECTORY` (e.g: `/run/lightsd`);
- lightsd needs a lightsd user, make sure it is created when the package gets installed and make sure the user is informed, e.g:

```
lightsd runs under the `lightsd' user and group by default; add yourself
to this group to be able to open lightsd's socket and pipe under
/run/lightsd:

  gpasswd -a $USER lightsd

Re-open your current desktop or ssh session for the change to take effect.
Then use systemctl to start lightsd; you can start playing with lightsd
with:

  `lightsd --prefix`/share/doc/lightsd/examples/lightsc.py
```

- make sure lightsd is built with hardening flags;
- make sure the protocol documentation and the examples are properly shipped.

Overall, I'm all in favor of a tight collaboration between up and downstream and have enough experience to package lightsd myself on pretty much any operating system; most of it will be automated with the release process down the road. What I really need is help to get lightsd in official distribution channels and repositories.

CHAPTER 7

Changelog

lightsd uses semantic versionning, here is the summary:

Given a version number MAJOR.MINOR.PATCH:

- MAJOR version gets incremented when you may need to modify your lightsd configuration to keep your setup running;
- MINOR version gets incremented when functionality or substantial improvements are added in a backwards-compatible manner;
- PATCH version gets incremented for backwards-compatible bug fixes.

7.1 1.1.2 (2015-11-30)

- Fix LIFX LAN protocol V2 handling (properly set RES_REQUIRED and properly listen on each gateway's socket);
- The bulb timeout has been increased from 3 to 20s;
- Improved LIFX traffic logging.

7.2 1.1.1 (2015-11-17)

Warning: This release broke the compatibility with the LIFX LAN protocol "v2", please upgrade to 1.1.2.

- Greatly improve responsiveness by setting the LIFX source identifier[1].
- Fix parallel builds in the Debian package & fix the homebrew formulae for OS X 10.11 (El Capitan).

[1] http://lan.developer.lifx.com/docs/header-description#frame

7.3 1.1.0 (2015-11-07)

Note: The `-f` (`--foreground`) option is being deprecated with this release and isn't documented anymore, lightsd starts in the foreground by default and this option is not necessary, please stop using it.

7.3.1 New features

- Add syslog support via the `--syslog` and `--syslog-facility` options (closes GH-1);
- Debian & OpenWRT packaging and installation instructions.

7.3.2 Fixes

- lightsc.sh: support OSes with openssl but without a base64 utility (closes GH-3);
- lightsc.py: unix url support fixes and bump the receive buffer size to accommodate people with many bulbs;
- Add missing product ids/models.

7.4 1.0.1 (2015-09-18)

- Fix set_waveform on big endian architectures;
- Fix build under Debian oldstable;
- Fix build under OpenBSD[2];
- Fix process title even when no bulbs are discovered;
- Add product id for the 230V version of the LIFX White 800.

7.5 1.0.0 (2015-09-17)

- First announced release.

[2] Using GCC 4.2, so you just need to do `pkg_add cmake libevent` to build a release.

CHAPTER 8

Indices and tables

- genindex
- search

Index