c-lightning

Release dd3d8d9-modded

January 21, 2020

User Documentation

CHAPTER 1

Install

1.1 Library Requirements

You will need several development libraries:

- libsqlite3: for database support.
- libgmp: for secp256k1
- zlib: for compression routines.

For actually doing development and running the tests, you will also need:

- pip3: to install python-bitcoinlib
- valgrind: for extra debugging checks

You will also need a version of bitcoind with segregated witness and `estimatesmartfee` economical node, such as the 0.16 or above.

1.2 To Build on Ubuntu

OS version: Ubuntu 15.10 or above

Get dependencies:

```
sudo apt-get update
sudo apt-get install -y \
  autoconf automake build-essential git libtool libgmp-dev \
  libsqlite3-dev python python3 python3-mako net-tools zlib1g-dev libsodium-dev \
  git gettext
```

If you don't have Bitcoin installed locally you'll need to install that as well:

```
sudo apt-get install software-properties-common
sudo add-apt-repository ppa:bitcoin/bitcoin
sudo apt-get update
sudo apt-get install -y bitcoind
```

Clone lightning:

```
git clone https://github.com/ElementsProject/lightning.git
cd lightning
```

For development or running tests, get additional dependencies:

```
sudo apt-get install -y valgrind python3-pip libpq-dev
sudo pip3 install -r tests/requirements.txt -r doc/requirements.txt
```

Build lightning:

```
./configure
make
sudo make install
```

Running lightning:

```
bitcoind &
./lightningd/lightningd &
./cli/lightning-cli help
```

Note: You may need to include `testnet=1` in `bitcoin.conf`

1.3 To Build on Fedora

OS version: Fedora 27 or above

Get dependencies:

```
$ sudo dnf update -y && \
        sudo dnf groupinstall -y \
                'C Development Tools and Libraries' \
                'Development Tools' && \
        sudo dnf install -y \
                clang \
```

(continues on next page)

(continued from previous page)

```
                gettext \
                git \
                gmp-devel \
                libsq3-devel \
                python2-devel \
                python3-devel \
                python3-pip \
                python3-setuptools \
                net-tools \
                valgrind \
                wget \
                zlib-devel \
                                libsodium-devel && \
        sudo dnf clean all
```

Make sure you have bitcoind available to run

Clone lightning:

```
$ git clone https://github.com/ElementsProject/lightning.git
$ cd lightning
```

Build and install lightning:

```
$lightning> ./configure
$lightning> make
$lightning> sudo make install
```

Running lightning (mainnet):

```
$ bitcoind &
$ lightningd --network=bitcoin
```

Running lightning on testnet:

```
$ bitcoind -testnet &
$ lightningd --network=testnet
```

1.4 To Build on FreeBSD

OS version: FreeBSD 11.1-RELEASE or above

c-lightning is in the FreeBSD ports, so install it as any other port (dependencies are handled automatically):

```
# pkg install c-lightning
```

for a binary, pre-compiled package. If you want to compile locally and fiddle with compile time options:

```
# cd /usr/ports/net-p2p/c-lightning && make install
```

mrkd is required to build man pages from markdown files (not done by the port):

```
# cd /usr/ports/devel/py-pip && make install
$ pip install --user mrkd
```

See `/usr/ports/net-p2p/c-lightning/Makefile` for instructions on how to build from an arbitrary git commit, instead of the latest release tag.

Note: Make sure you've set an utf-8 locale, e.g. `export LC_CTYPE=en_US.UTF-8`, otherwise manpage installation may fail.

Running lightning:

Configure bitcoind, if not already: add `rpcuser=<foo>` and `rpcpassword=<bar>` to `/usr/local/etc/bitcoin.conf`, maybe also `testnet=1`.

Configure lightningd: copy `/usr/local/etc/lightningd-bitcoin.conf.sample` to `/usr/local/etc/lightningd-bitcoin.conf` and edit according to your needs.

```
# service bitcoind start
# service lightningd start
# lightning-cli --rpc-file /var/db/c-lightning/bitcoin/lightning-rpc --lightning-dir=/
↪var/db/c-lightning help
```

1.5 To Build on NixOS

Use nix-shell launch a shell with a full clightning dev environment:

```
$ nix-shell -Q -p gdb sqlite autoconf git clang libtool gmp sqlite autoconf \
autogen automake libsodium 'python3.withPackages (p: [p.bitcoinlib])' \
valgrind --run make
```

1.6 To Build on macOS

Assuming you have Xcode and Homebrew installed. Install dependencies:

```
$ brew install autoconf automake libtool python3 gmp gnu-sed gettext libsodium
$ ln -s /usr/local/Cellar/gettext/0.20.1/bin/xgettext /usr/local/opt
$ export PATH="/usr/local/opt:$PATH"
```

If you need SQLite (or get a SQLite mismatch build error):

```
$ brew install sqlite
$ export LDFLAGS="-L/usr/local/opt/sqlite/lib"
$ export CPPFLAGS="-I/usr/local/opt/sqlite/include"
```

If you need Python 3.x for mako (or get a mako build error):

```
$ brew install pyenv
$ echo -e 'if command -v pyenv 1>/dev/null 2>&1; then\n  eval "$(pyenv init -)"\nfi' >
↪> ~/.bash_profile
$ source ~/.bash_profile
$ pyenv install 3.7.4
$ pip install --upgrade pip
```

If you don't have bitcoind installed locally you'll need to install that as well:

```
$ brew install berkeley-db4 boost miniupnpc pkg-config libevent
$ git clone https://github.com/bitcoin/bitcoin
$ cd bitcoin
$ ./autogen.sh
$ ./configure
$ make src/bitcoind src/bitcoin-cli && make install
```

Clone lightning:

```
$ git clone https://github.com/ElementsProject/lightning.git
$ cd lightning
```

Configure Python 3.x & get mako:

```
$ pyenv local 3.7.4
$ pip install mako
```

Build lightning:

```
$ ./configure
$ make
```

Running lightning:

Note: Edit your `~/Library/Application\ Support/Bitcoin/bitcoin.conf` to include `rpcuser=<foo>` and `rpcpassword=<bar>` first, you may also need to include `testnet=1`

```
bitcoind &
./lightningd/lightningd &
./cli/lightning-cli help
```

1.7 To cross-compile for Android

Make a standalone toolchain as per https://developer.android.com/ndk/guides/standalone_toolchain.html. For c-lightning you must target an API level of 24 or higher.

Depending on your toolchain location and target arch, source env variables such as:

```
export PATH=$PATH:/path/to/android/toolchain/bin
# Change next line depending on target device arch
target_host=arm-linux-androideabi
export AR=$target_host-ar
export AS=$target_host-clang
export CC=$target_host-clang
export CXX=$target_host-clang++
export LD=$target_host-ld
export STRIP=$target_host-strip
```

Two makefile targets should not be cross-compiled so we specify a native CC:

```
make CC=clang clean ccan/tools/configurator/configurator
make clean -C ccan/ccan/cdump/tools \
  && make CC=clang -C ccan/ccan/cdump/tools
```

Install the `qemu-user` package. This will allow you to properly configure the build for the target device environment. Build with:

```
BUILD=x86_64 MAKE_HOST=arm-linux-androideabi \
  make PIE=1 DEVELOPER=0 \
  CONFIGURATOR_CC="arm-linux-androideabi-clang -static"
```

1.8 To cross-compile for Raspberry Pi

Obtain the official Raspberry Pi toolchains. This document assumes compilation will occur towards the Raspberry Pi 3 (arm-linux-gnueabihf as of Mar. 2018).

Depending on your toolchain location and target arch, source env variables will need to be set. They can be set from the command line as such:

```
export PATH=$PATH:/path/to/arm-linux-gnueabihf/bin
# Change next line depending on specific Raspberry Pi device
target_host=arm-linux-gnueabihf
export AR=$target_host-ar
export AS=$target_host-as
export CC=$target_host-gcc
export CXX=$target_host-g++
export LD=$target_host-ld
export STRIP=$target_host-strip
```

Install the `qemu-user` package. This will allow you to properly configure the build for the target device environment. Config the arm elf interpreter prefix:

```
export QEMU_LD_PREFIX=/path/to/raspberry/arm-bcm2708/arm-rpi-4.9.3-linux-gnueabihf/
↪arm-linux-gnueabihf/sysroot/
```

Obtain and install cross-compiled versions of sqlite3, gmp and zlib:

Download and build zlib:

```
wget https://zlib.net/zlib-1.2.11.tar.gz
tar xvf zlib-1.2.11.tar.gz
cd zlib-1.2.11
./configure --prefix=$QEMU_LD_PREFIX
make
make install
```

Download and build sqlite3:

```
wget https://www.sqlite.org/2018/sqlite-src-3260000.zip
unzip sqlite-src-3260000.zip
cd sqlite-src-3260000
./configure --enable-static --disable-readline --disable-threadsafe --disable-load-
↪extension --host=$target_host --prefix=$QEMU_LD_PREFIX
make
make install
```

Download and build gmp:

```
wget https://gmplib.org/download/gmp/gmp-6.1.2.tar.xz
tar xvf gmp-6.1.2.tar.xz
cd gmp-6.1.2
./configure --disable-assembly --host=$target_host --prefix=$QEMU_LD_PREFIX
```

(continues on next page)

(continued from previous page)

```
make
make install
```

Then, build c-lightning with the following commands:

```
./configure
make
```

1.9 To compile for Armbian

For all the other Pi devices out there, consider using Armbian.

You can compile in `customize-image.sh` using the instructions for Ubuntu.

A working example that compiles both bitcoind and c-lightning for Armbian can be found here.

1.10 Additional steps

Go to README for more information how to create an address, add funds, connect to a node, etc.

CHAPTER 2

Setting up TOR with c-lightning

To use any Tor features with c-lightning you must have Tor installed and running.

```
sudo apt install tor
```

then `/etc/init.d/tor start` or `sudo systemctl start tor` depending on your system configuration.

Most default setting should be sufficient.

To keep a safe configuration for minimal harassment (See Tor FAQ) just check that this line is present in the Tor config file `/etc/tor/torrc`:

```
ExitPolicy reject *:* # no exits allowed
```

This does not affect c-lightning connect, listen, etc.. It will only prevent your node from becoming a Tor exit node. Only enable this if you are sure about the implications.

If you don't want to create .onion addresses this should be enough.

There are several ways by which a c-lightning node can accept or make connections over Tor.

The node can be reached over Tor by connecting to its .onion address.

To provide the node with a .onion address you can:

- create a **non-persistent** address with an auto service or
- create a **persistent** address with a hidden service.

2.1 Creation of an auto service for non-persistent .onion addresses

To provide the node a non-persistent .onion address it is necessary to access the Tor auto service. These types of addresses change each time the Tor service is restarted.

*NOTE:If the node is required to be reachable only by **persistent** .onion addresses, this part can be skipped and it is necessary to set up a hidden service with the steps outlined in the next section.*

To create and use the auto service follow these steps:

Edit the Tor config file `/etc/tor/torrc`

You can configure the service authenticated by cookie or by password:

2.1.1 Service authenticated by cookie

Add the following lines in the `/etc/tor/torrc` file:

```
ControlPort 9051
CookieAuthentication 1
CookieAuthFileGroupReadable 1
```

2.1.2 Service authenticated by password

Alternatively, you can set the authentication to the service with a password by following these steps:

1. Create a hash of your password with

```
tor --hash-password yourpassword
```

This returns a line like

`16:533E3963988E038560A8C4EE6BBEE8DB106B38F9C8A7F81FE38D2A3B1F`

1. put these lines in the `/etc/tor/torrc` file:

```
ControlPort 9051
HashedControlPassword 16:533E3963988E038560A8C4EE6BBEE8DB106B38F9C8A7F81FE38D2A3B1F
```

Save the file and restart the Tor service. In linux:

`/etc/init.d/tor restart` or `sudo systemctl start tor` depending on the configuration of your system.

The auto service is used by adding `--addr=autotor:127.0.0.1:9051` if you want the address to be public or `--bind-addr=autotor:127.0.0.1:9051` if you don't want to publish it.

In the case where the auto service is authenticated through a password, it will be necessary to add the option `--tor-service-password=yourpassword` (not the hash).

The created non-persistent .onion address will be shown by the `lightning-cli getinfo` command. The other nodes will be able to `connect` to this .onion address through the 9735 port.

2.2 Creation of a hidden service for a persistent .onion address

To have a persistent .onion address other nodes can connect to, it is necessary to set up a Tor Hidden Service.

NOTE: In the case where only non-persistent addresses are required, you don't have to create the hidden service and you can skip this part.

Add these lines in the `/etc/tor/torrc` file:

```
HiddenServiceDir /var/lib/tor/lightningd-service_v2/
HiddenServicePort 1234 127.0.0.1:9735
```

If you want to create a version 3 address, you must also add `HiddenServiceVersion 3` so the whole section will be:

```
HiddenServiceDir /var/lib/tor/lightningd-service_v3/
HiddenServiceVersion 3
HiddenServicePort 1234 127.0.0.1:9735
```

The hidden lightning service will be reachable at port 1234 (global port) of the .onion address, which will be created at the restart of the Tor service. Both types of addresses can coexist on the same node.

Save the file and restart the Tor service. In linux:

`/etc/init.d/tor restart` or `sudo systemctl start tor` depending on the configuration of your system.

You will find the newly created address with:

```
sudo cat /var/lib/tor/lightningd-service_v2/hostname
```

or

```
sudo cat /var/lib/tor/lightningd-service_v3/hostname
```

in the case of a version 3 Tor address.

Now you are able to create:

- Non-persistent version 2 .onion address via auto service (temp-v2)
- Persistent version 2 and version 3 .onion addresseses (v2 and v3).

Let's see how to use them.

2.3 What do we support

| Case # | IP Number | Tor address |Incoming / Outgoing Tor | | ——- | ———- | ——————- |——————- | 1 | Public | NO | Outgoing | | 2 | Public | v2 [1] | Incoming [4] | | 3 | Public | temp-v2 [2] | Incoming | | 4 | Not Announced | v2 | Incoming | | 5 | Not Announced | temp-v2 | Incoming | | 6 | Public | v3 [3] + temp-v2 | Incoming | | 7 | Not Announced | v3 + v2 + temp-v2 | Incoming | | 8 | Public | NO | Outcoing socks5 . |

NOTE:

1. v2: The Version 2 onion address is persistent across Tor service restarts. It is created when you create the *Tor Hidden Service*.
2. temp-v2: The Version 2 onion address changes at each restart of the Tor service. A non-persistent .onion address is generated by accessing an *auto service*.
3. All the v3 addresses referes to .onion addresses version 3.
4. In all the "Incoming" use case, the node can also make "Outgoing" Tor connections (connect to a .onion address) by adding the `--proxy=127.0.0.1:9050` option.

2.3.1 Case #1 c-lightning has a public IP address and no Tor hidden service address, but can connect to an onion address via a Tor socks 5 proxy.

Without a .onion address, the node won't be reachable through Tor by other nodes but it will always be able to `connect` to a Tor enabled node (outbound connections), passing the `connect` request through the Tor service socks5 proxy. When the Tor service starts it creates a socks5 proxy which is by default at the address 127.0.0.1:9050.

If the node is started with the option `--proxy=127.0.0.1:9050` the node will be always able to connect to nodes with .onion address through the socks5 proxy.

You can always add this option, also in the other use cases, to add outgoing Tor capabilities.

If you want to `connect` to nodes ONLY via the Tor proxy, you have to add the `--always-use-proxy=true` option.

You can announce your public IP address through the usual method:

```
--bind-addr=internalIPAddress:port --announce-addr=externalIpAddress
```

if the node is into an internal network

```
--addr=externalIpAddress
```

if the node is not inside an internal network.

TIP: If you are unsure which of the two is suitable for you, find your internal and external address and see if they match.

In linux:

Discover your external IP address with: `curl ipinfo.io/ip`

and your internal IP Address with: `ip route get 1 | awk '{print $NF;exit}'`

If they match you can use the `--addr` command line option.

2.3.2 Case #2 c-lightning has a public IP address and a fixed Tor hidden service address that is persistent, so that external users can connect to this node.

To have your external IP address and your .onion address announced, you use the

```
--bind-addr=yourInternalIPAddress:port --announce-addr=yourexternalIPAddress:port --
↪announce-addr=your.onionAddress:port`
```

option.

If you are not inside an internal network you can use

```
--addr=yourIPAddress:port --announce-addr=your.onionAddress:port
```

your.onionAddress is the one created with the Tor hidden service (*see above*). The port is the one indicated as the hidden service port. If the hidden service creation line is `HiddenServicePort 1234 127.0.0.1:9735` the .onion address will be reachable at the 1234 port (the global port).

It will be possible to connect to this node with:

```
lightning-cli connect nodeID .onionAddress globalPort
```

through Tor where .onion address is in the form `xxxxxxxxxxxxxxxxxxxxxxxxxx.onion`, Or

```
lightning-cli connect nodeID yourexternalIPAddress Port
```

through Clearnet.

2.3.3 Case #3 c-lightning has a public IP address and a non-persisten Tor service address

In this case other nodes can connect to you via Clearnet or Tor.

To announce your IP address to the network, you add:

```
--bind-addr=internalAddress:port --announce-addr=yourExternalIPAddress
```

or `--addr=yourExternalIPAddress`if you are NOT on an internal network.

To get your non-persistent Tor address, add `--addr=autotor:127.0.0.1:9051` if you want to announce it or `--bind-addr=autotor:127.0.0.1:9051` if you don't want to announce it.

If the auto service is protected by password (*see above*) it is necessary to specify it with the option `--tor-service-password=yourpassword` (not the hash).

You will obtain the generated non persisten .onion address by reading the results of the `lightning-cli getinfo` command. Other nodes will be able to connect to the .onion address through the 9735 port.

2.3.4 Case #4 c-lightning has no public IP address, but has a fixed Tor hidden service address that is persistent

Other nodes can connect to the announced .onion address created with the hidden service (*see above*).

In this case In the `lightningd` command line you will specify:

```
--bind-addr=yourInternalIPAddress:port --announce-addr=your.onionAddress:port
```

or `--addr=your.onionAddress:port` if you are NOT on an internal network.

2.3.5 Case #5 c-lightning has no public IP address, and has no fixed Tor hidden service address

In this case it is difficult to track the node. You specify just:

```
--bind-addr=yourInternalIPAddress:port --addr=autotor:127.0.0.1:9051
```

In the `lightningd` command line.

Other nodes will not be able to `connect` to you unless you communicate them how to reach you. You will find your .onion address with the command `lightning-cli getinfo` and the other nodes will be able to connect to it through the 9735 port.

2.3.6 Case #6 c-lightning has a public IP address and a fixed Tor V3 service address and a Tor V2 service address

You will be reachable via Clearnet, via Tor to the .onion V3 address and the .onion V2 address if this last is communicated to the node that wants to connect with our node.

to make your external IP address public you add:

```
--bind-addr=yourInternalAddress:port --announce-addr=yourexternalIPAddress:port`.
```

If the node is not on an internal network the option will be: `--addr=yourexternalIPAddress:port`.

Once the .onion addresses have been created with the procedures *oulined above*, the node is already reachable at the .onion address.

To make your external .onion addresses public you add:

```
--announce-addr=.onionAddressV2:port --announce-addr=.onionAddressV3:port
```

to the options to publish your IP number.

2.3.7 Case #7 c-lightning has no public IP address, a fixed Tor V3 service address, a fixed Tor V2 service address and also a 3rd non persisten V2 address

External users can connect to this node by Tor V2 and V3 and a random V2 until next tor release, then also (V3 randomly).

The Persistent addresses can be created with the steps *outlined above*.

To create your non-persistent Tor address, add `--addr=autotor:127.0.0.1:9051` if you want to announce it or `--bind-addr=autotor:127.0.0.1:9051` if you don't want to announce it.

Also you must specify `--tor-service-password=yourpassword` (not the hash) to access the Tor service at 9051 If you have protected them with the password (no additional options if they are protected with a cookie file. *See above*).

To make your external .onion address (V2 and V3) public you add:

```
--bind-addr=yourInternalIPAddress:port --announce-addr=your.onionAddressV2:port --
↪announce-addr=your.onionAddressV3:port
```

2.3.8 Case #8 c-lightning has a public IP address and no Tor addresses

The external address is communicated by the

```
--bind-addr=internalIPAddress:port --announce-addr=yourexternalIPAddress:port`
```

or `--addr=yourexternalIPAddress:port` if the node is not inside an internal network.

The node can connect to any V4/6 ip address via a IPV4/6 socks 5 proxy by specifing

```
--proxy=127.0.0.1:9050 --always-use-proxy=true
```

2.4 References

The Tor project

CHAPTER 3

Plugins

Plugins are a simple yet powerful way to extend the functionality provided by c-lightning. They are subprocesses that are started by the main `lightningd` daemon and can interact with `lightningd` in a variety of ways:

- **Command line option passthrough** allows plugins to register their own command line options that are exposed through `lightningd` so that only the main process needs to be configured.
- **JSON-RPC command passthrough** adds a way for plugins to add their own commands to the JSON-RPC interface.
- **Event stream subscriptions** provide plugins with a push-based notification mechanism about events from the `lightningd`.
- **Hooks** are a primitive that allows plugins to be notified about internal events in `lightningd` and alter its behavior or inject custom behaviors.

A plugin may be written in any language, and communicates with `lightningd` through the plugin's `stdin` and `stdout`. JSON-RPCv2 is used as protocol on top of the two streams, with the plugin acting as server and `lightningd` acting as client. The plugin file needs to be executable (e.g. use `chmod a+x plugin_name`)

3.1 A day in the life of a plugin

During startup of `lightningd` you can use the `--plugin=` option to register one or more plugins that should be started. In case you wish to start several plugins you have to use the `--plugin=` argument once for each plugin (or `--plugin-dir` or place them in the default plugin dirs, usually `/usr/local/libexec/c-lightning/plugins` and `~/.lightningd/plugins`). An example call might look like:

```
lightningd --plugin=/path/to/plugin1 --plugin=path/to/plugin2
```

`lightningd` will run your plugins from the `--lightning-dir`/networkname, then will write JSON-RPC requests to the plugin's `stdin` and will read replies from its `stdout`. To initialize the plugin two RPC methods are required:

- `getmanifest` asks the plugin for command line options and JSON-RPC commands that should be passed through. This can be run before `lightningd` checks that it is the sole user of the `lightning-dir` directory (for `--help`) so your plugin should not touch files at this point.
- `init` is called after the command line options have been parsed and passes them through with the real values (if specified). This is also the signal that `lightningd`'s JSON-RPC over Unix Socket is now up and ready to receive incoming requests from the plugin.

Once those two methods were called `lightningd` will start passing through incoming JSON-RPC commands that were registered and the plugin may interact with `lightningd` using the JSON-RPC over Unix-Socket interface.

3.1.1 The `getmanifest` method

The `getmanifest` method is required for all plugins and will be called on startup without any params. It MUST return a JSON object similar to this example:

```
{
  "options": [
    {
      "name": "greeting",
      "type": "string",
      "default": "World",
      "description": "What name should I call you?"
    }
  ],
  "rpcmethods": [
    {
      "name": "hello",
      "usage": "[name]",
      "description": "Returns a personalized greeting for {greeting} (set via␣
↪options)."
    },
    {
      "name": "gettime",
      "usage": "",
      "description": "Returns the current time in {timezone}",
      "long_description": "Returns the current time in the timezone that is given as␣
↪the only parameter.\nThis description may be quite long and is allowed to span␣
↪multiple lines."
    }
  ],
  "subscriptions": [
    "connect",
    "disconnect"
  ],
  "hooks": [
    "openchannel",
    "htlc_accepted"
  ],
  "dynamic": true
}
```

The `options` will be added to the list of command line options that `lightningd` accepts. The above will add a `--greeting` option with a default value of `World` and the specified description. *Notice that currently string, (unsigned) integers, and bool options are supported.*

The `rpcmethods` are methods that will be exposed via `lightningd`'s JSON-RPC over Unix-Socket interface, just like the builtin commands. Any parameters given to the JSON-RPC calls will be passed through verbatim. Notice that

the `name`, `description` and `usage` fields are mandatory, while the `long_description` can be omitted (it'll be set to `description` if it was not provided). `usage` should surround optional parameter names in `[]`.

The `dynamic` indicates if the plugin can be managed after `lightningd` has been started. Critical plugins that should not be stopped should set it to false.

Plugins are free to register any `name` for their `rpcmethod` as long as the name was not previously registered. This includes both built-in methods, such as `help` and `getinfo`, as well as methods registered by other plugins. If there is a conflict then `lightningd` will report an error and exit.

3.1.2 The `init` method

The `init` method is required so that `lightningd` can pass back the filled command line options and notify the plugin that `lightningd` is now ready to receive JSON-RPC commands. The `params` of the call are a simple JSON object containing the options:

```
{
  "options": {
    "greeting": "World"
  },
  "configuration": {
    "lightning-dir": "/home/user/.lightning/testnet",
    "rpc-file": "lightning-rpc",
    "startup": true
  }
}
```

The plugin must respond to `init` calls, however the response can be arbitrary and will currently be discarded by `lightningd`. JSON-RPC commands were chosen over notifications in order not to force plugins to implement notifications which are not that well supported.

The `startup` field allows a plugin to detect if it was started at `lightningd` startup (true), or at runtime (false).

3.2 JSON-RPC passthrough

Plugins may register their own JSON-RPC methods that are exposed through the JSON-RPC provided by `lightningd`. This provides users with a single interface to interact with, while allowing the addition of custom methods without having to modify the daemon itself.

JSON-RPC methods are registered as part of the `getmanifest` result. Each registered method must provide a `name` and a `description`. An optional `long_description` may also be provided. This information is then added to the internal dispatch table, and used to return the help text when using `lightning-cli help`, and the methods can be called using the `name`.

For example the above `getmanifest` result will register two methods, called `hello` and `gettime`:

```
  ...
  "rpcmethods": [
    {
      "name": "hello",
      "usage": "[name]",
      "description": "Returns a personalized greeting for {greeting} (set via␣
↪options)."
    },
    {
```

(continues on next page)

(continued from previous page)

```
      "name": "gettime",
      "description": "Returns the current time in {timezone}",
      "usage": "",
      "long_description": "Returns the current time in the timezone that is given as␣
↪the only parameter.\nThis description may be quite long and is allowed to span␣
↪multiple lines."
    }
  ],
  ...
```

The RPC call will be passed through unmodified, with the exception of the JSON-RPC call `id`, which is internally remapped to a unique integer instead, in order to avoid collisions. When passing the result back the `id` field is restored to its original value.

Note that if your `result` for an RPC call includes `"format-hint": "simple"`, then `lightning-cli` will default to printing your output in "human-readable" flat form.

3.3 Event notifications

Event notifications allow a plugin to subscribe to events in `lightningd`. `lightningd` will then send a push notification if an event matching the subscription occurred. A notification is defined in the JSON-RPC specification as an RPC call that does not include an `id` parameter:

> A Notification is a Request object without an "id" member. A Request object that is a Notification signifies the Client's lack of interest in the corresponding Response object, and as such no Response object needs to be returned to the client. The Server MUST NOT reply to a Notification, including those that are within a batch request.
>
> Notifications are not confirmable by definition, since they do not have a Response object to be returned. As such, the Client would not be aware of any errors (like e.g. "Invalid params","Internal error").

Plugins subscribe by returning an array of subscriptions as part of the `getmanifest` response. The result for the `getmanifest` call above for example subscribes to the two topics `connect` and `disconnect`. The topics that are currently defined and the corresponding payloads are listed below.

3.3.1 Notification Types

channel_opened

A notification for topic `channel_opened` is sent if a peer successfully funded a channel with us. It contains the peer id, the funding amount (in millisatoshis), the funding transaction id, and a boolean indicating if the funding transaction has been included into a block.

```
{
  "channel_opened": {
    "id": "03864ef025fde8fb587d989186ce6a4a186895ee44a926bfc370e2c366597a3f8f",
    "funding_satoshis": "100000000msat",
    "funding_txid": "4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b
↪",
    "funding_locked": false
  }
}
```

connect

A notification for topic `connect` is sent every time a new connection to a peer is established.

```
{
  "id": "02f6725f9c1c40333b67faea92fd211c183050f28df32cac3f9d69685fe9665432",
  "address": "1.2.3.4"
}
```

disconnect

A notification for topic `disconnect` is sent every time a connection to a peer was lost.

```
{
  "id": "02f6725f9c1c40333b67faea92fd211c183050f28df32cac3f9d69685fe9665432"
}
```

invoice_payment

A notification for topic `invoice_payment` is sent every time an invoie is paid.

```
{
  "invoice_payment": {
    "label": "unique-label-for-invoice",
    "preimage": "0000000000000000000000000000000000000000000000000000000000000000",
    "msat": "10000msat"
  }
}
```

warning

A notification for topic `warning` is sent every time a new `BROKEN` /`UNUSUAL` level(in plugins, we use `error`/`warn`) log generated, which means an unusual/borken thing happens, such as channel failed, message resolving failed...

```
{
  "warning": {
    "level": "warn",
    "time": "1559743608.565342521",
    "source": "lightningd(17652):␣
↪0821f80652fb840239df8dc99205792bba2e559a05469915804c08420230e23c7c chan #7854:",
    "log": "Peer permanent failure in CHANNELD_NORMAL: lightning_channeld: sent ERROR␣
↪bad reestablish dataloss msg"
  }
}
```

1. `level` is `warn` or `error`: `warn` means something seems bad happened and it's under control, but we'd better check it; `error` means something extremely bad is out of control, and it may lead to crash;
2. `time` is the second since epoch;
3. `source` means where the event happened, it may have the following forms: `<node_id> chan #<db_id_of_channel>:`,`lightningd(<lightningd_pid>):`, `plugin-<plugin_name>:`,

`<daemon_name>(<daemon_pid>):, jsonrpc:, jcon fd <error_fd_to_jsonrpc>:, plugin-manager;`

4. `log` is the context of the original log entry.

forward_event

A notification for topic `forward_event` is sent every time the status of a forward payment is set. The json format is same as the API `listforwards`.

```
{
  "forward_event": {
    "payment_hash": "f5a6a059a25d1e329d9b094aeeec8c2191ca037d3f5b0662e21ae850debe8ea2
↪",
    "in_channel": "103x2x1",
    "out_channel": "103x1x1",
    "in_msatoshi": 100001001,
    "in_msat": "100001001msat",
    "out_msatoshi": 100000000,
    "out_msat": "100000000msat",
    "fee": 1001,
    "fee_msat": "1001msat",
    "status": "settled",
    "received_time": 1560696342.368,
    "resolved_time": 1560696342.556
  }
}
```

or

```
{
  "forward_event": {
    "payment_hash": "ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
↪",
    "in_channel": "103x2x1",
    "out_channel": "110x1x0",
    "in_msatoshi": 100001001,
    "in_msat": "100001001msat",
    "out_msatoshi": 100000000,
    "out_msat": "100000000msat",
    "fee": 1001,
    "fee_msat": "1001msat",
    "status": "local_failed",
    "failcode": 16392,
    "failreason": "WIRE_PERMANENT_CHANNEL_FAILURE",
    "received_time": 1560696343.052
  }
}
```

- The status includes `offered`, `settled`, `failed` and `local_failed`, and they are all string type in json.
  - When the forward payment is valid for us, we'll set `offered` and send the forward payment to next hop to resolve;
  - When the payment forwarded by us gets paid eventually, the forward payment will change the status from `offered` to `settled`;
  - If payment fails locally(like failing to resolve locally) or the corresponding htlc with next hop fails(like htlc timeout), we will set the status as `local_failed`. `local_failed` may be set before setting

offered or after setting offered. In fact, from the time we receive the htlc of the previous hop, all we can know the cause of the failure is treated as local_failed. local_failed only occuors locally or happens in the htlc between us and next hop;

* If local_failed is set before offered, this means we just received htlc from the previous hop and haven't generate htlc for next hop. In this case, the json of forward_event sets the fields of out_msatoshi, out_msat,fee and out_channel as 0;
  - Note: In fact, for this case we may be not sure if this incoming htlc represents a pay to us or a payment we need to forward. We just simply treat all incoming failed to resolve as local_failed.
* Only in local_failed case, json includes failcode and failreason fields;

– failed means the payment forwarded by us fails in the latter hops, and the failure isn't related to us, so we aren't accessed to the fail reason. failed must be set after offered.

* failed case doesn't include failcode and failreason fields;

- received_time means when we received the htlc of this payment from the previous peer. It will be contained into all status case;
- resolved_time means when the htlc of this payment between us and the next peer was resolved. The resolved result may success or fail, so only settled and failed case contain resolved_time;
- The failcode and failreason are defined in BOLT 4.

sendpay_success

A notification for topic sendpay_success is sent every time a sendpay succeeds (with complete status). The json is the same as the return value of the commands sendpay/waitsendpay when these commands succeed.

```
{
  "sendpay_success": {
    "id": 1,
    "payment_hash": "5c85bf402b87d4860f4a728e2e58a2418bda92cd7aea0ce494f11670cfbfb206
↪",
    "destination": "035d2b1192dfba134e10e540875d366ebc8bc353d5aa766b80c090b39c3a5d885d
↪",
    "msatoshi": 100000000,
    "amount_msat": "100000000msat",
    "msatoshi_sent": 100001001,
    "amount_sent_msat": "100001001msat",
    "created_at": 1561390572,
    "status": "complete",
    "payment_preimage":
↪"9540d98095fd7f37687ebb7759e733934234d4f934e34433d4998a37de3733ee"
  }
}
```

sendpay doesn't wait for the result of sendpay and waitsendpay returns the result of sendpay in specified time or timeout, but sendpay_success will always return the result anytime when sendpay successes if is was subscribed.

sendpay_failure

A notification for topic sendpay_failure is sent every time a sendpay completes with failed status. The JSON is same as the return value of the commands sendpay/waitsendpay when these commands fail.

```
{
  "sendpay_failure": {
    "code": 204,
    "message": "failed: WIRE_UNKNOWN_NEXT_PEER (reply from remote)",
    "data": {
      "id": 2,
      "payment_hash":
↪"9036e3bdbd2515f1e653cb9f22f8e4c49b73aa2c36e937c926f43e33b8db8851",
      "destination":
↪"035d2b1192dfba134e10e540875d366ebc8bc353d5aa766b80c090b39c3a5d885d",
      "msatoshi": 100000000,
      "amount_msat": "100000000msat",
      "msatoshi_sent": 100001001,
      "amount_sent_msat": "100001001msat",
      "created_at": 1561395134,
      "status": "failed",
      "erring_index": 1,
      "failcode": 16394,
      "failcodename": "WIRE_UNKNOWN_NEXT_PEER",
      "erring_node":
↪"022d223620a359a47ff7f7ac447c85c46c923da53389221a0054c11c1e3ca31d59",
      "erring_channel": "103x2x1",
      "erring_direction": 0
    }
  }
}
```


`sendpay` doesn't wait for the result of sendpay and `waitsendpay` returns the result of sendpay in specified time or timeout, but `sendpay_failure` will always return the result anytime when sendpay fails if is was subscribed.

3.4 Hooks

Hooks allow a plugin to define custom behavior for `lightningd` without having to modify the c-lightning source code itself. A plugin declares that it'd like to be consulted on what to do next for certain events in the daemon. A hook can then decide how `lightningd` should react to the given event.

Hooks and notifications are very similar, however there are a few key differences:

- Notifications are asynchronous, i.e., `lightningd` will send the notifications but not wait for the plugin to process them. Hooks on the other hand are synchronous, `lightningd` cannot finish processing the event until the plugin has returned.
- Any number of plugins can subscribe to a notification topic, however only one plugin may register for any hook topic at any point in time (we cannot disambiguate between multiple plugins returning contradictory results from a hook callback).

Hooks are considered to be an advanced feature due to the fact that `lightningd` relies on the plugin to tell it what to do next. Use them carefully, and make sure your plugins always return a valid response to any hook invocation.

3.4.1 Hook Types

peer_connected

This hook is called whenever a peer has connected and successfully completed the cryptographic handshake. The parameters have the following structure if there is a channel with the peer:

```
{
  "peer": {
    "id": "03864ef025fde8fb587d989186ce6a4a186895ee44a926bfc370e2c366597a3f8f",
    "addr": "34.239.230.56:9735",
    "features": ""
  }
}
```

The hook is sparse on purpose, since the plugin can use the JSON-RPC `listpeers` command to get additional details should they be required. The `addr` field shows the address that we are connected to ourselves, not the gossiped list of known addresses. In particular this means that the port for incoming connections is an ephemeral port, that may not be available for reconnections.

The returned result must contain a `result` member which is either the string `disconnect` or `continue`. If `disconnect` and there's a member `error_message`, that member is sent to the peer before disconnection.

db_write

This hook is called whenever a change is about to be committed to the database. It is currently extremely restricted:

1. a plugin registering for this hook should not perform anything that may cause a db operation in response (pretty much, anything but logging).
2. a plugin registering for this hook should not register for other hooks or commands, as these may become intermingled and break rule #1.
3. the hook will be called before your plugin is initialized!

```
{
  "writes": [
    "PRAGMA foreign_keys = ON"
  ]
}
```

Any response but "true" will cause lightningd to error without committing to the database!

invoice_payment

This hook is called whenever a valid payment for an unpaid invoice has arrived.

```
{
  "payment": {
    "label": "unique-label-for-invoice",
    "preimage": "0000000000000000000000000000000000000000000000000000000000000000",
    "msat": "10000msat"
  }
}
```

The hook is sparse on purpose, since the plugin can use the JSON-RPC `listinvoices` command to get additional details about this invoice. It can return a non-zero `failure_code` field as defined for final nodes in BOLT 4, or otherwise an empty object to accept the payment.

openchannel

This hook is called whenever a remote peer tries to fund a channel to us, and it has passed basic sanity checks:

```
{
  "openchannel": {
    "id": "03864ef025fde8fb587d989186ce6a4a186895ee44a926bfc370e2c366597a3f8f",
    "funding_satoshis": "100000000msat",
    "push_msat": "0msat",
    "dust_limit_satoshis": "546000msat",
    "max_htlc_value_in_flight_msat": "18446744073709551615msat",
    "channel_reserve_satoshis": "1000000msat",
    "htlc_minimum_msat": "0msat",
    "feerate_per_kw": 7500,
    "to_self_delay": 5,
    "max_accepted_htlcs": 483,
    "channel_flags": 1
  }
}
```

There may be additional fields, including `shutdown_scriptpubkey` and a hex-string. You can see the definitions of these fields in BOLT 2's description of the open_channel message.

The returned result must contain a `result` member which is either the string `reject` or `continue`. If `reject` and there's a member `error_message`, that member is sent to the peer before disconnection.

For a 'continue'd result, you can also include a `close_to` address, which will be used as the output address for a mutual close transaction.

e.g.

```
{
    "result": "continue",
    "close_to": "bc1qlq8srqnz64wgklmqvurv7qnr4rvtq2u96hhfg2"
}
```

Note that `close_to` must be a valid address for the current chain; an invalid address will cause the node to exit with an error.

htlc_accepted

The `htlc_accepted` hook is called whenever an incoming HTLC is accepted, and its result determines how `lightningd` should treat that HTLC.

The payload of the hook call has the following format:

```
{
  "onion": {
    "payload": "",
    "type": "legacy",
    "short_channel_id": "1x2x3",
    "forward_amount": "42msat",
    "outgoing_cltv_value": 500014
  },
  "next_onion": "[1365bytes of serialized onion]",
  "shared_secret": "0000000000000000000000000000000000000000000000000000000000000000",
  "htlc": {
    "amount": "43msat",
    "cltv_expiry": 500028,
    "cltv_expiry_relative": 10,
    "payment_hash": "0000000000000000000000000000000000000000000000000000000000000000"
```

(continues on next page)

(continued from previous page)

```
  }
}
```

For detailed information about each field please refer to BOLT 04 of the specification, the following is just a brief summary:

- `onion.payload` contains the unparsed payload that was sent to us from the sender of the payment.
- `onion.type` is `legacy` for realm 0 payments, `tlv` for realm > 1.
- `short_channel_id` determines the channel that the sender is hinting should be used next. Not present if we're the final destination.
- `forward_amount` is the amount we should be forwarding to the next hop, and should match the incoming funds in case we are the recipient.
- `outgoing_cltv_value` determines what the CLTV value for the HTLC that we forward to the next hop should be.
- `total_msat` specifies the total amount to pay, if present.
- `payment_secret` specifies the payment secret (which the payer should have obtained from the invoice), if present.
- `next_onion` is the fully processed onion that we should be sending to the next hop as part of the outgoing HTLC. Processed in this case means that we took the incoming onion, decrypted it, extracted the payload destined for us, and serialized the resulting onion again.
- `shared_secret` is the shared secret we used to decrypt the incoming onion. It is shared with the sender that constructed the onion.
- `htlc`:
  - `amount` is the amount that we received with the HTLC. This amount minus the `forward_amount` is the fee that will stay with us.
  - `cltv_expiry` determines when the HTLC reverts back to the sender. `cltv_expiry` minus `outgoing_cltv_expiry` should be equal or larger than our `cltv_delta` setting.
  - `cltv_expiry_relative` hints how much time we still have to claim the HTLC. It is the `cltv_expiry` minus the current `blockheight` and is passed along mainly to avoid the plugin having to look up the current blockheight.
  - `payment_hash` is the hash whose `payment_preimage` will unlock the funds and allow us to claim the HTLC.

The hook response must have one of the following formats:

```
{
  "result": "continue"
}
```

This means that the plugin does not want to do anything special and `lightningd` should continue processing it normally, i.e., resolve the payment if we're the recipient, or attempt to forward it otherwise. Notice that the usual checks such as sufficient fees and CLTV deltas are still enforced.

```
{
  "result": "fail",
  "failure_code": 4301
}
```

`fail` will tell `lightningd` to fail the HTLC with a given numeric `failure_code` (please refer to the spec for details).

```
{
  "result": "resolve",
  "payment_key": "0000000000000000000000000000000000000000000000000000000000000000"
}
```

`resolve` instructs `lightningd` to claim the HTLC by providing the preimage matching the `payment_hash` presented in the call. Notice that the plugin must ensure that the `payment_key` really matches the `payment_hash` since `lightningd` will not check and the wrong value could result in the channel being closed.

Warning: `lightningd` will replay the HTLCs for which it doesn't have a final verdict during startup. This means that, if the plugin response wasn't processed before the HTLC was forwarded, failed, or resolved, then the plugin may see the same HTLC again during startup. It is therefore paramount that the plugin is idempotent if it talks to an external system.

rpc_command

The `rpc_command` hook allows a plugin to take over any RPC command. It sends the received JSON-RPC request to the registered plugin,

```
{
    "rpc_command": {
        "id": 3,
        "method": "method_name",
        "params": {
            "param_1": [],
            "param_2": {},
            "param_n": "",
        }
    }
}
```

which can in turn:

Let `lightningd` execute the command with

```
{
    "continue": true
}
```

Replace the request made to `lightningd`:

```
{
    "replace": {
        "id": 3,
        "method": "method_name",
        "params": {
            "param_1": [],
            "param_2": {},
            "param_n": "",
        }
    }
}
```

Return a custom response to the request sender:

```
{
    "return": {
        "result": {
        }
    }
}
```

Return a custom error to the request sender:

```
{
    "return": {
        "error": {
        }
    }
}
```

CHAPTER 4

Hacking

Welcome, fellow coder!

This repository contains a code to run a lightning protocol daemon. It's broken into subdaemons, with the idea being that we can add more layers of separation between different clients and extra barriers to exploits.

It is designed to implement the lightning protocol as specified in various BOLTs.

4.1 Getting Started

It's in C, to encourage alternate implementations. Patches are welcome! You should read our *Style Guide*.

To read the code, you should start from lightningd.c and hop your way through the '~' comments at the head of each daemon in the suggested order.

4.2 The Components

Here's a list of parts, with notes:

- ccan - useful routines from http://ccodearchive.net
  - Use make update-ccan to update it.
  - Use make update-ccan CCAN_NEW="mod1 mod2..." to add modules
  - Do not edit this! If you want a wrapper, add one to common/utils.h.
- bitcoin/ - bitcoin script, signature and transaction routines.
  - Not a complete set, but enough for our purposes.
- external/ - external libraries from other sources
  - libbacktrace - library to provide backtraces when things go wrong.
  - libsodium - encryption library (should be replaced soon with built-in)

  - libwally-core - bitcoin helper library
  - secp256k1 - bitcoin curve encryption library within libwally-core
  - jsmn - tiny JSON parsing helper
- tools/ - tools for building
  - check-bolt.c: check the source code contains correct BOLT quotes (as used by check-source)
  - generate-wire.py: generates wire marshal/unmarshal-ing routines for subdaemons and BOLT specs.
  - mockup.sh / update-mocks.sh: tools to generate mock functions for unit tests.
- tests/ - blackbox tests (mainly)
  - unit tests are in tests/ subdirectories in each other directory.
- doc/ - you are here
- devtools/ - tools for developers
  - Generally for decoding our formats.
- contrib/ - python support and other stuff which doesn't belong :)
- wire/ - basic marshalling/un for messages defined in the BOLTs
- common/ - routines needed by any two or more of the directories below
- cli/ - commandline utility to control lightning daemon.
- lightningd/ - master daemon which controls the subdaemons and passes peer file descriptors between them.
- wallet/ - database code used by master for tracking what's happening.
- hsmd/ - daemon which looks after the cryptographic secret, and performs commitment signing.
- gossipd/ - daemon to maintain routing information and broadcast gossip.
- connectd/ - daemon to connect to other peers, and receive incoming.
- openingd/ - daemon to open a channel for a single peer, and chat to a peer which doesn't have any channels/
- channeld/ - daemon to operate a single peer once channel is operating normally.
- closingd/ - daemon to handle mutual closing negotiation with a single peer.
- onchaind/ - daemon to handle a single channel which has had its funding transaction spent.

4.3 Debugging

You can build c-lightning with DEVELOPER=1 to use dev commands listed in `cli/lightning-cli help`. `./configure --enable-developer` will do that. You can log console messages with log_info() in lightningd and status_debug() in other subdaemons.

You can debug crashing subdaemons with the argument `--dev-debugger=channeld`, where `channeld` is the subdaemon name. It will run `gnome-terminal` by default with a gdb attached to the subdaemon when it starts. You can change the terminal used by setting the `DEBUG_TERM` environment variable, such as `DEBUG_TERM="xterm -e"` or `DEBUG_TERM="konsole -e"`.

It will also print out (to stderr) the gdb command for manual connection. The subdaemon will be stopped (it sends itself a SIGSTOP); you'll need to `continue` in gdb.

4.4 Database

c-lightning state is persisted in `lightning-dir`. It is a sqlite database stored in the `lightningd.sqlite3` file, typically under `~/.lightning/<network>/`. You can run queries against this file like so:

```
$ sqlite3 ~/.lightning/bitcoin/lightningd.sqlite3 \
  "SELECT HEX(prev_out_tx), prev_out_index, status FROM outputs"
```

Or you can launch into the sqlite3 repl and check things out from there:

```
$ sqlite3 ~/.lightning/bitcoin/lightningd.sqlite3
SQLite version 3.21.0 2017-10-24 18:55:49
Enter ".help" for usage hints.
sqlite> .tables
channel_configs  invoices         peers            vars
channel_htlcs    outputs          shachain_known   version
channels         payments         shachains
sqlite> .schema outputs
...
```

Some data is stored as raw bytes, use `HEX(column)` to pretty print these.

Make sure that clightning is not running when you query the database, as some queries may lock the database and cause crashes.

4.4.1 Common variables

Table `vars` contains global variables used by lightning node.

```
$ sqlite3 ~/.lightning/bitcoin/lightningd.sqlite3
SQLite version 3.21.0 2017-10-24 18:55:49
Enter ".help" for usage hints.
sqlite> .headers on
sqlite> select * from vars;
name|val
next_pay_index|2
bip32_max_index|4
...
```

Variables:

- `next_pay_index` next resolved invoice counter that will get assigned.
- `bip32_max_index` last wallet derivation counter.

Note: Each time `newaddr` command is called, `bip32_max_index` counter is increased to the last derivation index. Each address generated after `bip32_max_index` is not included as lightning funds.

4.5 Testing

Install `valgrind` and the python dependencies for best results:

```
sudo apt install valgrind cppcheck shellcheck
pip3 install -r tests/requirements.txt
```

Re-run `configure` for the python dependencies

```
./configure --enable-developer
```

Tests are run with: `make check [flags]` where the pertinent flags are:

```
DEVELOPER=[0|1] - developer mode increases test coverage
VALGRIND=[0|1]  - detects memory leaks during test execution but adds a significant␣
↪delay
PYTEST_PAR=n    - runs pytests in parallel
```

A modern desktop can build and run through all the tests in a couple of minutes with:

```
make -j12 full-check PYTEST_PAR=24 DEVELOPER=1 VALGRIND=0
```

Adjust `-j` and `PYTEST_PAR` accordingly for your hardware.

There are three kinds of tests:

- **source tests** - run by `make check-source`, looks for whitespace, header order, and checks formatted quotes from BOLTs if BOLTDIR exists.
- **unit tests** - standalone programs that can be run individually. You can also run all of the unit tests with `make check-units`. They are `run-*.c` files in test/ subdirectories used to test routines inside C source files.

  You should insert the lines when implementing a unit test:

  ```
  /* AUTOGENERATED MOCKS START */
  ```

  ```
  /* AUTOGENERATED MOCKS END */
  ```

  and `make update-mocks` will automatically generate stub functions which will allow you to link (and conveniently crash if they're called).
- **blackbox tests** - These tests setup a mini-regtest environment and test lightningd as a whole. They can be run individually:

  ```
  PYTHONPATH=contrib/pylightning:contrib/pyln-client:contrib/pyln-testing
  py.test -v tests/
  ```

  You can also append `-k TESTNAME` to run a single test. Environment variables `DEBUG_SUBD=<subdaemon>` and `TIMEOUT=<seconds>` can be useful for debugging subdaemons on individual tests.
- **pylightning tests** - will check contrib pylightning for codestyle and run the tests in `contrib/pylightning/tests` afterwards:

  ```
  make check-python
  ```

Our Travis CI instance (see `.travis.yml`) runs all these for each pull request.

4.6 Making BOLT Modifications

All of code for marshalling/unmarshalling BOLT protocol messages is generated directly from the spec. These are pegged to the BOLTVERSION, as specified in `Makefile`.

4.7 Source code analysis

An updated version of the NCC source code analysis tool is available at

https://github.com/bitonic-cjp/ncc

It can be used to analyze the lightningd source code by running `make clean && make ncc`. The output (which is built in parallel with the binaries) is stored in .nccout files. You can browse it, for instance, with a command like `nccnav lightningd/lightningd.nccout`.

4.8 Subtleties

There are a few subtleties you should be aware of as you modify deeper parts of the code:

- `ccan/structeq`'s STRUCTEQ_DEF will define safe comparison function foo_eq() for struct foo, failing the build if the structure has implied padding.
- `command_success`, `command_fail`, and `command_fail_detailed` will free the `cmd` you pass in. This also means that if you `tal`-allocated anything from the `cmd`, they will also get freed at those points and will no longer be accessible afterwards.
- When making a structure part of a list, you will instance a `struct list_node`. This has to be the *first* field of the structure, or else `dev-memleak` command will think your structure has leaked.

4.9 Protocol Modifications

The source tree contains CSV files extracted from the v1.0 BOLT specifications (wire/extracted_peer_wire_csv and wire/extracted_onion_wire_csv). You can regenerate these by setting `BOLTDIR` and `BOLTVERSION` appropriately, and running `make extract-bolt-csv`.

4.10 Further Information

Feel free to ask questions on the lightning-dev mailing list, or on `#c-lightning` on IRC, or email me at rusty@rustcorp.com.au.

Cheers! Rusty.

CHAPTER 5

Care And Feeding of Your Fellow Coders

Style is an individualistic thing, but working on software is group activity, so consistency is important. Generally our coding style is similar to the Linux coding style.

5.1 Communication

We communicate with each other via code; we polish each others code, and give nuanced feedback. Exceptions to the rules below always exist: accept them. Particularly if they're funny!

5.2 Prefer Short Names

`num_foos` is better than `number_of_foos`, and `i` is better than `counter`. But `bool found;` is better than `bool ret;`. Be as short as you can but still descriptive.

5.3 Prefer 80 Columns

We have to stop somewhere. The two tools here are extracting deeply-indented code into their own functions, and use of short-cuts using early returns or continues, eg:

```
        for (i = start; i != end; i++) {
                if (i->something)
                        continue;

                if (!i->something_else)
                        continue;

                do_something(i);
}
```

5.4 Prefer Simple Statements

Notice the statement above uses separate tests, rather than combining them. We prefer to only combine conditionals which are fundamentally related, eg:

```
        if (i->something != NULL && *i->something < 100)
```

5.5 Use of `take()`

Some functions have parameters marked with `TAKES`, indicating that they can take lifetime ownership of a parameter which is passed using `take()`. This can be a useful optimization which allows the function to avoid making a copy, but if you hand `take(foo)` to something which doesn't support `take()` you'll probably leak memory!

In particular, our automatically generated marshalling code doesn't support `take()`.

If you're allocating something simply to hand it via `take()` you should use NULL as the parent for clarity, eg:

```
        msg = towire_shutdown(NULL, &peer->channel_id, peer->final_scriptpubkey);
        enqueue_peer_msg(peer, take(msg));
```

5.6 Use of `tmpctx`

There's a convenient temporary tal context which gets cleaned regularly: you should use this for throwaways rather than (as you'll see some of our older code do!) grabbing some passing object to hang your temporaries off!

5.7 Enums and Switch Statements

If you handle various enumerated values in a `switch`, don't use `default:` but instead mention every enumeration case-by-case. That way when a new enumeration case is added, most compilers will warn that you don't cover it. This is particularly valuable for code auto-generated from the specification!

5.8 Initialization of Variables

Avoid double-initialization of variables; it's better to set them when they're known, eg:

```
        bool is_foo;

        if (bar == foo)
                is_foo = true;
        else
                is_foo = false;

        ...
        if (is_foo)...
```

This way the compiler will warn you if you have one path which doesn't set the variable. If you initialize with `bool is_foo = false;` then you'll simply get that value without warning when you change the code and forget to set it on one path.

5.9 Initialization of Memory

`valgrind` warns about decisions made on uninitialized memory. Prefer `tal` and `tal_arr` to `talz` and `tal_arrz` for this reason, and initialize only the fields you expect to be used.

Similarly, you can use `memcheck(mem, len)` to explicitly assert that memory should have been initialized, rather than having valgrind trigger later. We use this when placing things on queues, for example.

5.10 Use of static and const

Everything should be declared static and const by default. Note that `tal_free()` can free a const pointer (also, that it returns `NULL`, for convenience).

5.11 Typesafety Is Worth Some Pain

If code is typesafe, refactoring is as simple as changing a type and compiling to find where to refactor. We rely on this, so most places in the code will break if you hand the wrong type, eg `type_to_string` and `structeq`.

The two tools we have to help us are complicated macros in `ccan/typesafe_cb` allow you to create callbacks which must match the type of their argument, rather than using `void *`. The other is `ARRAY_SIZE`, a macro which won't compile if you hand it a pointer instead of an actual array.

5.12 Use of `FIXME`

There are two cases in which you should use a `/* FIXME: */` comment: one is where an optimization is possible but it's not clear that it's yet worthwhile, and the second one is to note an ugly corner case which could be improved (and may be in a following patch).

There are always compromises in code: eventually it needs to ship. `FIXME` is `grep`-fodder for yourself and others, as well as useful warning signs if we later encounter an issue in some part of the code.

5.13 If You Don't Know The Right Thing, Do The Simplest Thing

Sometimes the right way is unclear, so it's best not to spend time on it. It's far easier to rewrite simple code than complex code, too.

5.14 Write For Today: Unused Code Is Buggy Code

Don't overdesign: complexity is a killer. If you need a fancy data structure, start with a brute force linked list. Once that's working, perhaps consider your fancy structure, but don't implement a generic thing. Use `/* FIXME: ...*/` to salve your conscience.

5.15 Keep Your Patches Reviewable

Try to make a single change at a time. It's tempting to do "drive-by" fixes as you see other things, and a minimal amount is unavoidable, but you can end up shaving infinite yaks. This is a good time to drop a `/* FIXME: ...*/` comment and move on.

5.16 Github Workflows

We have adopted a number of workflows to facilitate the development of c-lightning, and to make things more pleasant for contributors.

5.16.1 Changelog Entries in Commit Messages

We are maintaining a chanelog in the top-level directory of this project. However since every pull request has a tendency to touch the file and therefore create merge-conflicts we decided to derive the changelog file from the pull requests that were added between releases. In order for a pull request to show up in the changelog at least one of its commits will have to have a line with one of the following prefixes:

- `Changelog-Added:` if the pull request adds a new feature
- `Changelog-Changed:` if a feature has been modified and might require changes on the user side
- `Changelog-Deprecated:` if a feature has been marked for deprecation, but not yet removed
- `Changelog-Fixed:` if a bug has been fixed
- `Changelog-Removed:` if a (previously deprecated) feature has been removed
- `Changelog-Security:` if a security issue has been addressed and the users will need to upgrade in order to stay secure

In case you think the pull request is small enough not to require a changelog entry please use `Changelog-None` in one of the commit messages to opt out.

Under some circumstances a feature may be removed even without deprecation warning if it was not part of a released version yet, or the removal is urgent.

In order to ensure that each pull request has the required `Changelog-*:` line for the changelog our trusty @bitcoin-bot will check logs whenever a pull request is created or updated and search for the required line. If there is no such line it'll mark the pull request as `pending` to call out the need for an entry.

CHAPTER 6

Release checklist

Here's a checklist for the release process.

6.1 Leading Up To The Release

1. Talk to team about whether there are any changes which MUST go in this release which may cause delay.
2. Look through outstanding issues, to identify any problems that might be necessary to fixup before the release. Good candidates are reports of the project not building on different architectures or crashes.
3. Identify a good lead for each outstanding issue, and ask them about a fix timeline.
4. Create a milestone for the *next* release on Github, and go though open issues and PRs and mark accordingly.
5. Ask (via email) the most significant contributor who has not already named a release to name the release (use devtools/credit to find this contributor). CC previous namers and team.

6.2 Preparing for -rc1

1. Check that `CHANGELOG.md` is well formatted, ordered in areas, covers all signficant changes, and sub-ordered approximately by user impact & coolness.
2. Use `devtools/changelog.py` to collect the changelog entries from pull request commit messages and merge them into the manually maintained `CHANGELOG.md`.
3. Update the CHANGELOG.md with [Unreleased] changed to v-rc1. Note that you should exactly copy the date and name format from a previous release, as the `build-release.sh` script relies on this.
4. Create a PR with the above.

6.3 Releasing -rc1

1. Merge the above PR.
2. Tag it `git pull && git tag -s v<VERSION>rc1`. Note that you should get a prompt to give this tag a 'message'. Make sure you fill this in.
3. Confirm that the tag will show up for builds with `git describe`
4. Push the tag to remote `git push --tags`.
5. Update the /topic on #c-lightning on Freenode.
6. Prepare draft release notes (see devtools/credit), and share with team for editing.
7. Upgrade your personal nodes to the rc1, to help testing.
8. Test `tools/build-release.sh` to build the non-reprodicible images and reproducible zipfile.
9. Use the zipfile to produce a reproducible build.

6.4 Releasing -rc2, etc

1. Change rc1 to rc2 in CHANGELOG.md.
2. Add a PR with the rc2.
3. Tag it `git pull && git tag -s v<VERSION>rc2 && git push --tags`
4. Update the /topic on #c-lightning on Freenode.
5. Upgrade your personal nodes to the rc2.

6.5 Tagging the Release

1. Update the CHANGELOG.md; remove -rcN in both places, update the date, and add an [Unreleased] footnote URL from this new version to HEAD.
2. Add a PR with that release.
3. Merge the PR, then `git pull && git tag -s v<VERSION> && git push --tags`.
4. Run `tools/build-release.sh` to build the non-reprodicible images and reproducible zipfile.
5. Use the zipfile to produce a reproducible build.
6. Create the checksums for signing: `sha256sum release/* > release/SHA256SUMS`
7. Create the first signature with `gpg -sb --armor release/SHA256SUMS`
8. Upload the files resulting files to github and save as a draft. (https://github.com/ElementsProject/lightning/releases/)
9. Ping the rest of the team to check the SHA256SUMS file and have them send their `gpg -sb --armor SHA256SUMS`.
10. Append the signatures into a file called `SHA256SUMS.asc`, verify with `gpg --verify SHA256SUMS.asc` and include the file in the draft release.

6.6 Performing the Release

1. Edit the GitHub draft and include the `SHA256SUMS.asc` file.
2. Publish the release as not a draft.
3. Update the /topic on #c-lightning on Freenode.
4. Send a mail to c-lightning and lightning-dev mailing lists, using the same wording as the Release Notes in github.

6.7 Post-release

1. Add a new '[Unreleased]' section to the CHANGELOG.md with empty headers.
2. Look through PRs which were delayed for release and merge them.
3. Close out the Milestone for the now-shipped release.
4. Update this file with any missing or changed instructions.

CHAPTER 7

Changelog

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog and this project adheres to Semantic Versioning.

7.1 [0.8.0] - 2019-12-16: "Blockchain Good, Orange Coin Bad"

This release was named by Michael Schmoock @m-schmoock.

7.1.1 Added

- JSON API: Added `createonion` and `sendonion` JSON-RPC methods allowing the implementation of custom protocol extensions that are not directly implemented in c-lightning itself. (3260)
- JSON API: `listinvoices` now displays the payment preimage if the invoice was paid. (3295)
- JSON API: `listpeers` channels now include `close_to` and `close_to_addr` iff a `close_to` address was specified at channel open (3223)
- The new `pyln-testing` package now contains the testing infrastructure so it can be reused to test against c-lightning in external projects (3218)
- config: configuration files now support `include`. (3268)
- options: Allow the Tor inbound service port differ from 9735 (3155)
- options: Persistent Tor address support (3155)
- plugins: A new plugin hook, `rpc_command` allows a plugin to take over `lightningd` for any RPC command. (2925)
- plugins: Allow the `accepter` to specify an upfront_shutdown_script for a channel via a `close_to` field in the openchannel hook result (3280)
- plugins: Plugins may now handle modern TLV-style payloads via the `htlc_accepted` hook (3260)

- plugins: libplugin now supports writing plugins which register to hooks (3317)
- plugins: libplugin now supports writing plugins which register to notifications (3317)
- protocol: Payment amount fuzzing is restored, but through shadow routing. (3212)
- protocol: We now signal the network we are running on at init. (3300)
- protocol: can now send and receive TLV-style onion messages. (3335)
- protocol: can now send and receive BOLT11 payment_secrets. (3335)
- protocol: can now receive basic multi-part payments. (3335)
- JSON RPC: low-level commands sendpay and waitsendpay can now be used to manually send multi-part payments. (3335)
- quirks: Workaround LND's `reply_channel_range` issues instead of sending error. (3264)
- tools: A new command, `guesstoremote`, is added to the hsmtool. It is meant to be used to recover funds after an unilateral close of a channel with `option_static_remotekey` enabled. (3292)

7.1.2 Changed

:warning: The default network and the default location of the lightning home directory changed. Please make sure that the configuration, key file and database are moved into the network-specific subdirectory.

- config: Default network (new installs) is now bitcoin, not testnet. (3268)
- config: Lightning directory, plugins and files moved into `<network>/` subdir (3268)
- JSON API: The `fundchannel` command now tries to connect to the peer before funding the channel, no need to `connect` before `fundchannel` if an address for the peer is known (3314)
- JSON API: `htlc_accepted` hook has `type` (currently `legacy` or `tlv`) and other fields directly inside `onion`. (3167)
- JSON API: `lightning_` prefixes removed from subdaemon names, including in listpeers `owner` field. (3241)
- JSON API: `listconfigs` now structures plugins and include their options (3283)
- JSON API: the `raw_payload` now includes the first byte, i.e., the realm byte, of the payload as well. This allows correct decoding of a TLV payload in the plugins. (3261)
- logging: formatting made uniform: [NODEID-]SUBSYSTEM: MESSAGE (3241)
- options: `config` and `<network>/config` read by default. (3268)
- options: log-level can now specify different levels for different subsystems. (3241)
- protocol: The TLV payloads for the onion packets are no longer considered an experimental feature and generally available. (3260)
- quirks: We'll now reconnect and retry if we get an error on an established channel. This works around lnd sending error messages that may be non-fatal. (3340)

:warning: If you don't have a config file, you now may need to specify the network to `lightning-cli` (3268)

7.1.3 Deprecated

Note: You should always set `allow-deprecated-apis=false` to test for changes.

- JSON API: `listconfigs` duplicated "plugin" paths (3283)

- JSON API: `htlc_accepted` hook `per_hop_v0` object deprecated, as is `short_channel_id` for the final hop. (3167)

7.1.4 Removed

- JSON: `listpays` won't shown payments made via sendpay without a bolt11 string, or before 0.7.0. (3309)

7.1.5 Fixed

- JSON API: #3231 `listtransactions` crash (3256)
- JSON API: `listconfigs` appends '...' to truncated config options. (3268)
- `pyln-client` now handles unicode characters in JSON-RPC requests and responses correctly. (3018)
- bitcoin: If bitcoind goes backwards (e.g. reindex) refuse to start (unless forced with –rescan). (3274)
- bug: `gossipd` crash on huge number of unknown channels. (3273)
- gossip: No longer discard most `node_announcements` (fixes #3194) (3262)
- options: We disable all dns even on startup the scan for bogus dns servers, if `--always-use-proxy` is set true (3251)
- protocol: "Bad commitment signature" closing channels when we sent back-to-back update_fee messages across multiple reconnects. (3329)
- protocol: Unlikely corner case is simultanous HTLCs near balance limits fixed. (3286)

7.1.6 Security

7.2 0.7.3 - 2019-10-18: "Bitcoin's Proof of Stake"

This release was named by @trueptolemy.

7.2.1 Added

- DB: lightningd now supports different SQL backends, instead of the default which is sqlite3. Adds a PostgresSQL driver
- elements: Add support of Liquid-BTC on elements
- JSON API: `close` now accepts an optional parameter `destination`, to which the to-local output will be sent.
- JSON API: `txprepare` and `withdraw` now accept an optional parameter `utxos`, a list of utxos to include in the prepared transaction
- JSON API: `listfunds` now lists a blockheight for confirmed transactions, and has `connected` and `state` fields for channels, like `listpeers`.
- JSON API: `fundchannel_start` now includes field `scriptpubkey`
- JSON API: New method `listtransactions`
- JSON API: `signmessage` will now create a signature from your node on a message; `checkmessage` will verify it.

- JSON API: `fundchannel_start` now accepts an optional parameter `close_to`, the address to which these channel funds should be sent to on close. Returns `using_close_to` if will use.
- Plugin: new notifications `sendpay_success` and `sendpay_failure`.
- Protocol: nodes now announce features in `node_announcement` broadcasts.
- Protocol: we now offer `option_gossip_queries_ex` for finegrained gossip control.
- Protocol: we now retransmit `funding_locked` upon reconnection while closing if there was no update
- Protocol: no longer ask for `initial_routing_sync` (only affects ancient peers).
- bolt11: support for parsing feature bits (field `9`).
- Wallet: we now support the encryption of the BIP32 master seed (a.k.a. `hsm_secret`).
- pylightning: includes implementation of handshake protocol

7.2.2 Changed

- Build: Now requires `gettext`
- JSON API: The parameter `exclude` of `getroute` now also support node-id.
- JSON API: `txprepare` now uses `outputs` as parameter other than `destination` and `satoshi`
- JSON API: `fundchannel_cancel` is extended to work before funding broadcast.
- JSON-API: `pay` can exclude error nodes if the failcode of `sendpay` has the NODE bit set
- JSON API: The `plugin` command now returns on error. A timeout of 20 seconds is added to `start` and `startdir` subcommands at the end of which the plugin is errored if it did not complete the handshake with `lightningd`.
- JSON API: The `plugin` command does not allow to start static plugins after `lightningd` startup anymore.
- Protocol: We now push our own gossip to all peers, independent of their filter.
- Protocol: Now follows spec in responses to short channel id queries on unknown chainhashes
- Tor: We default now with autotor to generate if possible temporary ED25519-V3 onions. You can use new option `enable-autotor-v2-mode` to fallback to V2 RSA1024 mode.

7.2.3 Deprecated

Note: You should always set `allow-deprecated-apis=false` to test for changes.

- JSON API: `fundchannel` now uses `amount` as the parameter name to replace `satoshi`
- JSON API: `fundchannel_start` now uses `amount` as the parameter name to replace `satoshi`
- JSON API: `listpeers` and `listnodes` fields `localfeatures` and `globalfeatures` (now just `features`).
- Plugin: `peer_connected` hook fields `localfeatures` and `globalfeatures` (now just `features`).

7.2.4 Removed

- JSON API: `short_channel_id` parameters in JSON commands with `:` separators (deprecated since 0.7.0).
- JSON API: `description` parameters in `pay` and `sendpay` (deprecated since 0.7.0).

- JSON API: `description` output field in `waitsendpay` and `sendpay` (deprecated since 0.7.0).
- JSON API: `listpayments` (deprecated since 0.7.0).

7.2.5 Fixed

- Fixed bogus "Bad commit_sig signature" which caused channel closures when reconnecting after updating fees under simultaneous bidirectional traffic.
- Relative `--lightning_dir` is now working again.
- Build: MacOS now builds again (missing pwritev).

7.2.6 Security

7.3 0.7.2.1 - 2019-08-19: "Nakamoto's Pre-approval by US Congress"

This release was named by Antoine Poinsot @darosior.

(Technically a .1 release, as it contains last-minute fixes after 0.7.2 was tagged)

7.3.1 Added

- JSON API: a new command `plugin` allows one to manage plugins without restarting `lightningd`.
- Plugin: a new boolean field can be added to a plugin manifest, `dynamic`. It allows a plugin to tell if it can be started or stopped "on-the-fly".
- Plugin: a new boolean field is added to the `init`'s `configuration`, `startup`. It allows a plugin to know if it has been started on `lightningd` startup.
- Plugin: new notifications `invoice_payment`, `forward_event` and `channel_opened`.
- Protocol: `--enable-experimental-features` adds gossip query extensions aka https://github.com/lightningnetwork/lightning-rfc/pull/557
- contrib: new `bootstrap-node.sh` to connect to random mainnet nodes.
- JSON API: `listfunds` now returns also `funding_output` for `channels`
- Plugin: plugins can now suggest `lightning-cli` default to -H for responses.
- Lightningd: add support for `signet` networks using the `--network=signet` or `--signet` startup option

7.3.2 Changed

- Build: now requires `python3-mako` to be installed, i.e. `sudo apt-get install python3-mako`
- JSON API: `close` optional arguments have changed: it now defaults to unilateral close after 48 hours.
- Plugin: if the config directory has a `plugins` subdirectory, those are loaded.
- lightningd: check bitcoind version when setup topology and confirm the version not older than v0.15.0.
- Protocol: space out reconnections on startup if we have more than 5 peers.
- JSON API: `listforwards` includes the 'payment_hash' field.
- Plugin: now plugins always run from the `lightning-dir` for easy local storage.

7.3.3 Deprecated

Note: You should always set `allow-deprecated-apis=false` to test for changes.

- Plugin: using startup-relative paths for `plugin` and `plugin-dir`: they're now relative to `lightning-dir`.
- JSON API: `listforwards` removed dummy (zero) fields for `out_msat`, `fee_msat`, `in_channel` and `out_channel` if unknown (i.e. deleted from db, or `status` is `local-failed`.

7.3.4 Removed

7.3.5 Fixed

- Plugin: `pay` no longer crashes on timeout.
- Plugin: `disconnect` notifier now called if remote side disconnects.
- channeld: ignore, and simply try reconnecting if lnd sends "sync error".
- Protocol: we now correctly ignore unknown odd messages.
- wallet: We will now backfill blocks below our wallet start height on demand when we require them to verify gossip messages. This fixes an issue where we would not remove channels on spend that were opened below that start height because we weren't tracking the funding output.
- Detect when we're still syncing with bitcoin network: don't send or receive HTLCs or allow `fundchannel`.
- Rare onchaind error where we don't recover our own unilateral close with multiple same-preimage HTLCs fixed.

7.3.6 Security

7.4 0.7.1 - 2019-06-29: "The Unfailing Twitter Consensus Algorithm"

This release was named by (C-Lightning Core Team member) Lisa Neigut @niftynei.

7.4.1 Added

- Protocol: we now enforce `option_upfront_shutdown_script` if a peer negotiates it.
- JSON API: New command `setchannelfee` sets channel specific routing fees.
- JSON API: new withdraw methods `txprepare`, `txsend` and `txdiscard`.
- JSON API: add three new RPC commands: `fundchannel_start`, `fundchannel_complete` and `fundchannel_cancel`. Allows a user to initiate and complete a channel open using funds that are in a external wallet.
- Plugin: new hooks `db_write` for intercepting database writes, `invoice_payment` for intercepting invoices before they're paid, `openchannel` for intercepting channel opens, and `htlc_accepted` to decide whether to resolve, reject or continue an incoming or forwarded payment..
- Plugin: new notification `warning` to report any `LOG_UNUSUAL`/`LOG_BROKEN` level event.
- Plugin: Added a default plugin directory : `lightning_dir/plugins`. Each plugin directory it contains will be added to lightningd on startup.
- Plugin: the `connected` hook can now send an `error_message` to the rejected peer.

- JSON API: `newaddr` outputs `bech32` or `p2sh-segwit`, or both with new `all` parameter (#2390)
- JSON API: `listpeers` status now shows how many confirmations until channel is open (#2405)
- Config: Adds parameter `min-capacity-sat` to reject tiny channels.
- JSON API: `listforwards` now includes the time an HTLC was received and when it was resolved. Both are expressed as UNIX timestamps to facilitate parsing (Issue #2491, PR #2528)
- JSON API: `listforwards` now includes the local_failed forwards with failcode (Issue #2435, PR #2524)
- DB: Store the signatures of channel announcement sent from remote peer into DB, and init channel with signatures from DB directly when reenable the channel. (Issue #2409)
- JSON API: `listchannels` has new fields `htlc_minimum_msat` and `htlc_maximum_msat`.

7.4.2 Changed

- Gossip: we no longer compact the `gossip_store` file dynamically, due to lingering bugs. Restart if it gets too large.
- Protocol: no longer ask for entire gossip flood from peers, unless we're missing gossip.
- JSON API: `invoice` expiry defaults to 7 days, and can have s/m/h/d/w suffixes.
- Config: Increased default amount for minimal channel capacity from 1k sat to 10k sat.
- JSON API: A new parameter is added to `fundchannel`, which now accepts an utxo array to use to fund the channel.
- Build: Non-developer builds are now done with "-Og" optimization.
- JSON API: `pay` will no longer return failure until it is no longer retrying; previously it could "timeout" but still make the payment.
- JSON API: the command objects that `help` outputs now contain a new string field : `category` (can be "bitcoin", "channels", "network", "payment", "plugins", "utility", "developer" for native commands, or any other new category set by a plugin).
- Plugin: a plugin can now set the category of a newly created RPC command. This possibility has been added to libplugin.c and pylightning.
- lightning-cli: the human readable help is now more human and more readable : commands are sorted alphabetically and ordered by categories.

7.4.3 Deprecated

Note: You should always set `allow-deprecated-apis=false` to test for changes.

- JSON API: `newaddr` output field `address`: use `bech32` or `p2sh-segwit` instead.

7.4.4 Removed

- JSON RPC: `global_features` and `local_features` fields and `listchannels`' `flags` field. (Deprecated since 0.6.2).
- pylightning: Remove RPC support for c-lightning before 0.6.3.

7.4.5 Fixed

- Protocol: reconnection during closing negotiation now supports `option_data_loss_protect` properly.
- `--bind-addr=<path>` fixed for nodes using local sockets (eg. testing).
- Unannounced local channels were forgotten for routing on restart until reconnection occurred.
- lightning-cli: arguments containing `"` now succeed, rather than causing JSON errors.
- Protocol: handle lnd sending more messages before `reestablish`; don't fail channel, and handle older lnd's spurious empty commitments.
- Fixed `fundchannel` crash when we have many UTXOs and we skip unconfirmed ones.
- lightningd: fixed occasional hang on `connect` when peer had sent error.
- JSON RPC: `decodeinvoice` and `pay` now handle unknown invoice fields properly.
- JSON API: `waitsendpay` (PAY_STOPPED_RETRYING) error handler now returns valid JSON
- protocol: don't send multiple identical feerate changes if we want the feerate higher than we can afford.
- JSON API: `stop` now only returns once lightningd has released all resources.

7.4.6 Security

7.5 0.7.0 - 2019-02-28: "Actually an Altcoin"

This release was named by Mark Beckwith @wythe.

7.5.1 Added

- plugins: fully enabled, and ready for you to write some!
- plugins: `pay` is now a plugin.
- protocol: `pay` will now use routehints in invoices if it needs to.
- build: reproducible source zipfile and Ubuntu 18.04.1 build.
- JSON API: New command `paystatus` gives detailed information on `pay` commands.
- JSON API: `getroute`, `invoice`, `sendpay` and `pay` commands `msatoshi` parameter can have suffixes `msat`, `sat` (optionally with 3 decimals) or `btc` (with 1 to 11 decimal places).
- JSON API: `fundchannel` and `withdraw` commands `satoshi` parameter can have suffixes `msat` (must end in `000`), `sat` or `btc` (with 1 to 8 decimal places).
- JSON API: `decodepay`, `getroute`, `sendpay`, `pay`, `listpeers`, `listfunds`, `listchannels` and all invoice commands now return an `amount_msat` field which has an `msat` suffix.
- JSON API: `listfunds channels` now has `_msat` fields for each existing raw amount field, with `msat` suffix.
- JSON API: `waitsendpay` now has an `erring_direction` field.
- JSON API: `listpeers` now has a `direction` field in `channels`.
- JSON API: `listchannels` now takes a `source` option to filter by node id.
- JSON API: `getroute riskfactor` argument is simplified; `pay` now defaults to setting it to 10.

- JSON API: `sendpay` now takes a `bolt11` field, and it's returned in `listpayments` and `waitsendpay`.
- JSON API: `fundchannel` and `withdraw` now have a new parameter `minconf` that limits coinselection to outputs that have at least `minconf` confirmations (default 1). (#2380)
- JSON API: `listfunds` now displays addresses for all outputs owned by the wallet (#2387)
- JSON API: `waitsendpay` and `sendpay` output field `label` as specified by `sendpay` call.
- JSON API: `listpays` command for higher-level payment view than `listpayments`, especially important with multi-part-payments coming.
- JSON API: `listpayments` is now `listsendpays`.
- lightning-cli: `help <cmd>` finds man pages even if `make install` not run.
- pylightning: New class 'Millisatoshi' can be used for JSON API, and new '_msat' fields are turned into this on reading.

7.5.2 Changed

- protocol: `option_data_loss_protect` is now enabled by default.
- JSON API: The `short_channel_id` separator has been changed to be `x` to match the specification.
- JSON API: `listpeers` now includes `funding_allocation_msat`, which returns a map of the amounts initially funded to the channel by each peer, indexed by channel id.
- JSON API: `help` with a `command` argument gives a JSON array, like other commands.
- JSON API: `sendpay` `description` parameter is renamed `label`.
- JSON API: `pay` now takes an optional `label` parameter for labelling payments, in place of never-used `description`.
- build: we'll use the system libbase58 and libsodium if found suitable.

7.5.3 Deprecated

Note: You should always set `allow-deprecated-apis=false` to test for changes.

We recommend that you transition to the reading the new JSON `_msat` fields for your own sanity checking, and that you similarly provide appropriate suffixes for JSON input fields.

- JSON API: `short_channel_id` fields in JSON commands with `:` separators (use `x` instead).
- JSON API: `pay` `description` is deprecated, as is support for BOLT11 strings using `h`.
- JSON API: `sendpay` parameter `description` and `waitsendpay` and `sendpay` output fields `description` (now `label`).
- JSON API: `listpayments` has been deprecated (you probably want `listpays`)

7.5.4 Removed

- JSON API: the `waitsendpay` command error return no longer includes `channel_update`

7.5.5 Fixed

- Protocol: handling `query_channel_range` for large numbers of blocks (eg. 4 billion) was slow due to a bug.
- Fixed occasional deadlock with peers when exchanging huge amounts of gossip.
- Fixed a crash when running in daemon-mode due to db filename overrun (#2348)
- Handle lnd sending premature 'funding_locked' message when we're expected 'reestablish'; we used to close channel if this happened.
- Cleanup peers that started opening a channel, but then disconnected. These would leave a dangling entry in the DB that would cause this peer to be unable to connect. (PR #2371)
- You can no longer make giant unpayable "wumbo" invoices.
- CLTV of total route now correctly evaluated when finding best route.
- `riskfactor` arguments to `pay` and `getroute` now have an effect.
- Fixed the version of bip32 private_key to BIP32_VER_MAIN_PRIVATE: we used BIP32_VER_MAIN_PRIVATE for bitcoin/litecoin mainnet, and BIP32_VER_TEST_PRIVATE for others. (PR #2436)

7.5.6 Security

7.6 0.6.3 - 2019-01-09: "The Smallblock Conspiracy"

This release was named by @molxyz and @ctrlbreak.

7.6.1 Added

- JSON API: New command `check` checks the validity of a JSON API call without running it.
- JSON API: `getinfo` now returns `num_peers` `num_pending_channels`, `num_active_channels` and `num_inactive_channels` fields.
- JSON API: use `\n\n` to terminate responses, for simplified parsing (pylightning now relies on this)
- JSON API: `fundchannel` now includes an `announce` option, when false it will keep channel private. Defaults to true.
- JSON API: `listpeers`'s `channels` now includes a `private` flag to indicate if channel is announced or not.
- JSON API: `invoice` route hints may now include private channels if you have no public ones, unless new option `exposeprivatechannels` is false.
- Plugins: experimental plugin support for `lightningd`, including option passthrough and JSON-RPC passthrough.
- Protocol: we now support features `option_static_remotekey` and `gossip_queries_ex` for peers.

7.6.2 Changed

- JSON API: `pay` and `decodepay` accept and ignore `lightning:` prefixes.
- pylightning: Allow either keyword arguments or positional arguments.
- JSON-RPC: messages are now separated by 2 consecutive newlines.
- JSON-RPC: `jsonrpc:2.0` now included in json-rpc command calls. complies with spec.

7.6.3 Deprecated

Note: You should always set `allow-deprecated-apis=false` to test for changes.

- pylightning: Support for pre-2-newline JSON-RPC (<= 0.6.2 lightningd) is deprecated.

7.6.4 Removed

- option_data_loss_protect is now only offered if EXPERIMENTAL_FEATURES is enabled, since it seems incompatible with lnd and has known bugs.

7.6.5 Fixed

- JSON API: uppercase invoices now parsed correctly (broken in 0.6.2).
- JSON API: commands are once again read even if one hasn't responded yet (broken in 0.6.2).
- Protocol: allow lnd to send `update_fee` before `funding_locked`.
- Protocol: fix limit on how much funder can send (fee was 1000x too small)
- Protocol: don't send invalid onion errors if peer says onion was bad.
- Protocol: don't crash when peer sends a 0-block-expiry HTLC.
- pylightning: handle multiple simultanous RPC replies reliably.
- build: we use `--prefix` as handed to `./configure`

7.6.6 Security

7.7 0.6.2 - 2018-10-20: "The Consensus Loving Nasal Daemon"

This release was named by practicalswift.

7.7.1 Added

- JSON API: `listpeers` has new field `scratch_txid`: the latest tx in channel.
- JSON API: `listpeers` has new array `htlcs`: the current live payments.
- JSON API: `listchannels` has two new fields: `message_flags` and `channel_flags`. This replaces `flags`.
- JSON API: `invoice` now adds route hint to invoices for incoming capacity (RouteBoost), and warns if insufficient capacity.

- JSON API: `listforwards` lists all forwarded payments, their associated channels, and fees.
- JSON API: `getinfo` shows forwarding fees earnt as `msatoshi_fees_collected`.
- Bitcoind: more parallelism in requests, for very slow nodes.
- Testing: fixed logging, cleaner interception of bitcoind, minor fixes.
- Protocol: we set and handle the new `htlc_maximum_msat` channel_update field.

7.7.2 Changed

- Protocol: `channel_update` sent to disable channel only if we reject an HTLC.
- Protocol: we don't send redundant `node_announcement` on every new channel.
- Config: config file can override `lightning-dir` (makes sense with `--conf`).
- Config: `--conf` option is now relative to current directory, not `lightning-dir`.
- lightning-cli: `help <cmd>` prints basic information even if no man page found.
- JSON API: `getinfo` now reports global statistics about forwarded payments, including total fees earned and amounts routed.

7.7.3 Deprecated

Note: You should always set `allow-deprecated-apis=false` to test for changes.

- JSON RPC: `listchannels`' `flags` field. This has been split into two fields, see Added.
- JSON RPC: `global_features` and `local_features` fields: use `globalfeatures` and `localfeatures` as per BOLT #1.

7.7.4 Removed

- JSON API: the optional 'seed' parameter to `getroute` was removed.

7.7.5 Fixed

- Startup: more coherent complaint if daemon already running.
- Lightningd: correctly save full HTLCs across restarts; fixup old databases.
- JSON RPC: `getinfo` now shows correct Tor port.
- JSON RPC: `ping` now works even after one peer fails to respond.
- JSON RPC: `getroute` `fuzzpercent` and `pay` `maxfeepercent` can now be > 100.
- JSON RPC: `riskfactor` in `pay` and `getroute` no longer always treated as 1.
- JSON-RPC: `listpeers` was always reporting 0 for all stats.
- JSON RPC: `withdraw all` says `Cannot afford transaction` if you have absolutely no funds, rather than `Output 0 satoshis would be dust`.
- Protocol: don't send gossip about closed channels.
- Protocol: fix occasional deadlock when both peers flood with gossip.

- Protocol: fix occasional long delay on sending `reply_short_channel_ids_end`.
- Protocol: re-send `node_announcement` when address/alias/color etc change.
- Protocol: multiple HTLCs with the same payment_hash are handled correctly.
- Options: 'autotor' defaults to port 9051 if not specified.

7.7.6 Security

7.8 0.6.1 - 2018-09-11: "Principled Opposition To Segwit"

This release was named by ZmnSCPxj.

7.8.1 Added

- Protocol: gossipd now deliberately delays spamming with `channel_update`.
- Protocol: liveness ping when we commit changes but peer is idle: speeds up failures and reduces forced closures.
- Protocol: `option_data_loss_protect` now supported to protect peers against being out-of-date.
- JSON API: Added description to invoices and payments (#1740).
- JSON API: `getinfo` has new fields `alias` and `color`.
- JSON API: `listpeers` has new fields `global_features` and `local_features`.
- JSON API: `listnodes` has new field `global_features`.
- JSON API: `ping` command to send a ping to a connected peer.
- JSON API: `feerates` command to retrieve current fee estimates.
- JSON API: `withdraw` and `fundchannel` can be given manual feerate.
- Config: `--conf` option to set config file.
- Documentation: Added CHANGELOG.md
- pylightning: RpcError now has `method` and `payload` fields.
- Sending lightningd a SIGHUP will make it reopen its `log-file`, if any.

7.8.2 Changed

- Protocol: Fee estimates are now smoothed over time, to avoid sudden jumps.
- Config: You can only announce one address if each type (IPv4, IPv6, TORv2, TORv3).
- lightning-cli: the help command for a specific command now runs the `man` command.
- HSM: The HSM daemon now maintains the per-peer secrets, rather than handing them out. It's still lax in what it signs though.
- connectd: A new daemon `lightning_connectd` handles connecting to/from peers, instead of `gossipd` doing that itself. `lightning_openingd` now handles peers immediately, even if they never actually open a channel.
- Test: `python-xdist` is now a dependency for tests.

- Logging: JSON connections no longer spam debug logs.
- Routing: We no longer consider channels that are not usable either because of their capacity or their `htlc_minimum_msat` parameter (#1777)
- We now try to connect to all known addresses for a peer, not just the one given or the first one announced.
- Crash logs are now placed one-per file like `crash.log.20180822233752`
- We will no longer allow withdrawing funds or funding channels if we do not have a fee estimate (eg. bitcoind not synced); use new `feerate` arg.

7.8.3 Deprecated

7.8.4 Removed

- JSON API: `listpeers` results no long have `alias` and `color` fields; they're in `listnodes` (we used to internally merge the information).
- JSON API: `listpeers` will never have `state` field (it accidentally used to exist and set to `GOSSIPING` before we opened a channel). `connected` will indicate if we're connected, and the `channels` array indicates individual channel states (if any).
- Config: `default-fee-rate` is no longer available; use explicit `feerate` option if necessary.
- Removed all Deprecated options from 0.6.

7.8.5 Fixed

- Protocol: `node_announcement` multiple addresses are correctly ordered and uniquified.
- Protocol: if we can't estimate feerate, be almost infinitely tolerant of other side setting fees to avoid unilateral close.
- JSON API: `listnodes`: now displays node aliases and colors even if they don't advertise a network address
- JSON API: `fundchannel all`: now restricts to 2^24-1 satoshis rather than failing.
- JSON API: `listnodes`: now correctly prints `addresses` if more than one is advertised.
- Config: `bind-addr` of a publicly accessible network address was announced.
- When we reconnect and have to retransmit failing HTLCs, the errors weren't encrypted by us.
- `lightningd_config` man page is now installed by `make install`.
- Fixed crash when shutting down during opening a channel (#1737)
- Don't lose track of our own output when applying penalty transaction (#1738)
- Protocol: `channel_update` inside error messages now refers to correct channel.
- Stripping type prefix from `channel_updates` that are nested in an onion reply to be compatible with eclair and lnd (#1730).
- Failing tests no longer delete the test directory, to allow easier debugging (Issue: #1599)

7.8.6 Security

7.9 0.6 - 2018-06-22: "I Accidentally The Smart Contract"

In the prehistory of c-lightning, no changelog was kept. But major JSON API changes are tracked.

This release was named by Fabrice Drouin.

7.9.1 Deprecated

Note: You should always set `allow-deprecated-apis=false` to test for changes.

- Config: `port`. Use `addr=:<portnum>`.
- Config: `ipaddr`. Use `addr`.
- Config: `anchor-confirms`. Use `funding-confirms`.
- Config: `locktime-blocks`. Use `watchtime-blocks`.
- Protocol: on closing we allow out-of-range offers, prior to spec fix 2018-01-30 ("BOLT 2: order closing-signed negotiation by making funder send first." `90241d9cf60a598eac8fd839ac81e4093a161272`)
- JSON API: `listinvoice` command. Use `listinvoices`.
- JSON API: invoice result fields `paid_timestamp` and `expiry_time`. Use `paid_at` and `expires_at`.
- JSON API: `invoice` command field `fallback`. Use `fallbacks`.
- JSON API: `decodepay` result fields `timestamp` and `fallback`. Use `created_at` and `fallbacks`.
- JSON API: payment result fields `timestamp`. Use `created_at`.
- JSON API: `getinfo` result field `port`. Use `binding` and `address` arrays.
- JSON API: `getlog` result field `creation_time`. Use `created_at`.
- JSON API: `getpeers` result field `channel_reserve_satoshis`. Use `their_channel_reserve_satoshis`.
- JSON API: `getpeers` result field `to_self_delay`. Use `their_to_self_delay`.

7.10 Older versions

There predate the BOLT specifications, and are only of vague historic interest:

1. 0.1 - 2015-08-08: "MtGox's Cold Wallet" (named by Rusty Russell)
2. 0.2 - 2016-01-22: "Butterfly Labs' Timely Delivery" (named by Anthony Towns)
3. 0.3 - 2016-05-25: "Nakamoto's Genesis Coins" (named by Braydon Fuller)
4. 0.4 - 2016-08-19: "Wright's Cryptographic Proof" (named by Christian Decker)
5. 0.5 - 2016-10-19: "Bitcoin Savings & Trust Daily Interest" (named by Glenn Willen)
6. 0.5.1 - 2016-10-21
7. 0.5.2 - 2016-11-21: "Bitcoin Savings & Trust Daily Interest II"

CHAPTER 8

lightningd-config – Lightning daemon configuration file

8.1 SYNOPSIS

~/.lightning/config

8.2 DESCRIPTION

When lightningd(8) starts up it usually reads a general configuration file (default: **$HOME/.lightning/config**) then a network-specific configuration file (default: **$HOME/.lightning/testnet/config**). This can be changed: see *–conf* and *–lightning-dir*.

General configuration files are processed first, then network-specific ones, then command line options: later options override earlier ones except *addr* options and *log-level* with subsystems, which accumulate.

*include * followed by a filename includes another configuration file at that point, relative to the current configuration file.

All these options are mirrored as commandline arguments to lightningd(8), so *–foo* becomes simply *foo* in the configuration file, and *–foo=bar* becomes *foo=bar* in the configuration file.

Blank lines and lines beginning with # are ignored.

8.3 DEBUGGING

–help will show you the defaults for many options; they vary with network settings so you can specify *–network* before *–help* to see the defaults for that network.

The lightning-listconfigs(7) command will output a valid configuration file using the current settings.

8.4 OPTIONS

8.4.1 General options

allow-deprecated-apis=*BOOL* Enable deprecated options, JSONRPC commands, fields, etc. It defaults to *true*, but you should set it to *false* when testing to ensure that an upgrade won't break your configuration.

help Print help and exit. Not very useful inside a configuration file, but fun to put in other's config files while their computer is unattended.

version Print version and exit. Also useless inside a configuration file, but putting this in someone's config file may convince them to read this man page.

Bitcoin control options:

network=*NETWORK* Select the network parameters (*bitcoin*, *testnet*, or *regtest*). This is not valid within the per-network configuration file.

testnet Alias for *network=testnet*.

signet Alias for *network=signet*.

mainnet Alias for *network=bitcoin*.

bitcoin-cli=*PATH* The name of *bitcoin-cli* executable to run.

bitcoin-datadir=*DIR* *-datadir* argument to supply to bitcoin-cli(1).

bitcoin-rpcuser=*USER* The RPC username for talking to bitcoind(1).

bitcoin-rpcpassword=*PASSWORD* The RPC password for talking to bitcoind(1).

bitcoin-rpcconnect=*HOST* The bitcoind(1) RPC host to connect to.

bitcoin-rpcport=*PORT* The bitcoind(1) RPC port to connect to.

bitcoin-retry-timeout=*SECONDS* Number of seconds to keep trying a bitcoin-cli(1) command. If the command keeps failing after this time, exit with a fatal error.

rescan=*BLOCKS* Number of blocks to rescan from the current head, or absolute blockheight if negative. This is only needed if something goes badly wrong.

8.4.2 Lightning daemon options

lightning-dir=*DIR* Sets the working directory. All files (except *–conf* and *–lightning-dir* on the command line) are relative to this. This is only valid on the command-line, or in a configuration file specified by *–conf*.

pid-file=*PATH* Specify pid file to write to.

log-level=*LEVEL*[:*SUBSYSTEM*] What log level to print out: options are io, debug, info, unusual, broken. If *SUBSYSTEM* is supplied, this sets the logging level for any subsystem containing that string. Subsystems include:

- *lightningd*: The main lightning daemon
- *database*: The database subsystem
- *wallet*: The wallet subsystem
- *gossipd*: The gossip daemon
- *plugin-manager*: The plugin subsystem
- *plugin-P*: Each plugin, P = plugin path without directory

- *hsmd*: The secret-holding daemon
- *connectd*: The network connection daemon
- *jsonrpc#FD*: Each JSONRPC connection, FD = file descriptor number

  The following subsystems exist for each channel, where N is an incrementing internal integer id assigned for the lifetime of the channel:

- *openingd-chan#N*: Each opening / idling daemon
- *channeld-chan#N*: Each channel management daemon
- *closingd-chan#N*: Each closing negotiation daemon
- *onchaind-chan#N*: Each onchain close handling daemon

  So, **log-level=debug:plugin** would set debug level logging on all plugins and the plugin manager. **log-level=io:chan#55** would set IO logging on channel number 55 (or 550, for that matter).

log-prefix=*PREFIX* Prefix for log lines: this can be customized if you want to merge logs with multiple daemons.

log-file=*PATH* Log to this file instead of stdout. Sending lightningd(8) SIGHUP will cause it to reopen this file (useful for log rotation).

rpc-file=*PATH* Set JSON-RPC socket (or /dev/tty), such as for lightning-cli(1).

daemon Run in the background, suppress stdout and stderr.

conf=*PATH* Sets configuration file, and disable reading the normal general and network ones. If this is a relative path, it is relative to the starting directory, not **lightning-dir** (unlike other paths). *PATH* must exist and be readable (we allow missing files in the default case). Using this inside a configuration file is invalid.

wallet=*DSN* Identify the location of the wallet. This is a fully qualified data source name, including a scheme such as `sqlite3` or `postgres` followed by the connection parameters.

encrypted-hsm If set, you will be prompted to enter a password used to encrypt the `hsm_secret`. Note that once you encrypt the `hsm_secret` this option will be mandatory for `lightningd` to start.

8.4.3 Lightning node customization options

alias=*NAME* Up to 32 bytes of UTF-8 characters to tag your node. Completely silly, since anyone can call their node anything they want. The default is an NSA-style codename derived from your public key, but "Peter Todd" and "VAULTERO" are good options, too.

rgb=*RRGGBB* Your favorite color as a hex code.

fee-base=*MILLISATOSHI* Default: 1000. The base fee to charge for every payment which passes through. Note that millisatoshis are a very, very small unit! Changing this value will only affect new channels and not existing ones. If you want to change fees for existing channels, use the RPC call lightning-setchannelfee(7).

fee-per-satoshi=*MILLIONTHS* Default: 10 (0.001%). This is the proportional fee to charge for every payment which passes through. As percentages are too coarse, it's in millionths, so 10000 is 1%, 1000 is 0.1%. Changing this value will only affect new channels and not existing ones. If you want to change fees for existing channels, use the RPC call lightning-setchannelfee(7).

min-capacity-sat=*SATOSHI* Default: 10000. This value defines the minimal effective channel capacity in satoshi to accept for channel opening requests. If a peer tries to open a channel smaller than this, the opening will be rejected.

ignore-fee-limits=*BOOL* Allow nodes which establish channels to us to set any fee they want. This may result in a channel which cannot be closed, should fees increase, but make channels far more reliable since we never close it due to unreasonable fees.

commit-time=*MILLISECONDS* How long to wait before sending commitment messages to the peer: in theory increasing this would reduce load, but your node would have to be extremely busy node for you to even notice.

8.4.4 Lightning channel and HTLC options

watchtime-blocks=*BLOCKS* How long we need to spot an outdated close attempt: on opening a channel we tell our peer that this is how long they'll have to wait if they perform a unilateral close.

max-locktime-blocks=*BLOCKS* The longest our funds can be delayed (ie. the longest **watchtime-blocks** our peer can ask for, and also the longest HTLC timeout we will accept). If our peer asks for longer, we'll refuse to create a channel, and if an HTLC asks for longer, we'll refuse it.

funding-confirms=*BLOCKS* Confirmations required for the funding transaction when the other side opens a channel before the channel is usable.

commit-fee=*PERCENT* The percentage of *estimatesmartfee 2* to use for the bitcoin transaction which funds a channel: can be greater than 100.

commit-fee-min=*PERCENT* **commit-fee-max**=*PERCENT* Limits on what onchain fee range we'll allow when a node opens a channel with us, as a percentage of *estimatesmartfee 2*. If they're outside this range, we abort their opening attempt. Note that **commit-fee-max** can (should!) be greater than 100.

max-concurrent-htlcs=*INTEGER* Number of HTLCs one channel can handle concurrently in each direction. Should be between 1 and 483 (default 30).

cltv-delta=*BLOCKS* The number of blocks between incoming payments and outgoing payments: this needs to be enough to make sure that if we have to, we can close the outgoing payment before the incoming, or redeem the incoming once the outgoing is redeemed.

cltv-final=*BLOCKS* The number of blocks to allow for payments we receive: if we have to, we might need to redeem this on-chain, so this is the number of blocks we have to do that.

Invoice control options:

autocleaninvoice-cycle=*SECONDS* Perform cleanup of expired invoices every *SECONDS* seconds, or disable if 0. Usually unpaid expired invoices are uninteresting, and just take up space in the database.

autocleaninvoice-expired-by=*SECONDS* Control how long invoices must have been expired before they are cleaned (if *autocleaninvoice-cycle* is non-zero).

8.4.5 Networking options

Note that for simple setups, the implicit *autolisten* option does the right thing: it will try to bind to port 9735 on IPv4 and IPv6, and will announce it to peers if it seems like a public address.

You can instead use *addr* to override this (eg. to change the port), or precisely control where to bind and what to announce with the *bind-addr* and *announce-addr* options. These will **disable** the *autolisten* logic, so you must specifiy exactly what you want!

addr=*[IPADDRESS[:PORT]]|autotor:TORIPADDRESS[:SERVICEPORT][/torport=TORPORT]|statictor:TORIPADDRESS[:SERVIC*

Set an IP address (v4 or v6) or automatic Tor address to listen on and (maybe) announce as our node address.

```
An empty 'IPADDRESS' is a special value meaning bind to IPv4 and/or
IPv6 on all interfaces, '0.0.0.0' means bind to all IPv4
interfaces, '::' means 'bind to all IPv6 interfaces'.  If 'PORT' is
not specified, 9735 is used.  If we can determine a public IP
address from the resulting binding, and no other addresses of the
same type are already announced, the address is announced.
```

(continues on next page)

(continued from previous page)

```
If the argument begins with 'autotor:' then it is followed by the
IPv4 or IPv6 address of the Tor control port (default port 9051),
and this will be used to configure a Tor hidden service for port
9735.  The Tor hidden service will be configured to point to the
first IPv4 or IPv6 address we bind to.

If the argument begins with 'statictor:' then it is followed by the
IPv4 or IPv6 address of the Tor control port (default port 9051),
and this will be used to configure a static Tor hidden service for port
9735.  The Tor hidden service will be configured to point to the
first IPv4 or IPv6 address we bind to and is by default unique to
your nodes id. You can add the text '/torblob=BLOB' followed by up to
64 Bytes of text to generate from this text a v3 onion service
address text unique to the first 32 Byte of this text.
You can also use an postfix '/torport=TORPORT' to select the external
tor binding. The result is that over tor your node is accessible by a port
defined by you and possible different from your local node port assignment

This option can be used multiple times to add more addresses, and
its use disables autolisten.  If necessary, and 'always-use-proxy'
is not specified, a DNS lookup may be done to resolve 'IPADDRESS'
or 'TORIPADDRESS'.
```

bind-addr=*[IPADDRESS[:PORT]]|SOCKETPATH* Set an IP address or UNIX domain socket to listen to, but do not announce. A UNIX domain socket is distinguished from an IP address by beginning with a */*.

```
An empty 'IPADDRESS' is a special value meaning bind to IPv4 and/or
IPv6 on all interfaces, '0.0.0.0' means bind to all IPv4
interfaces, '::' means 'bind to all IPv6 interfaces'.  'PORT' is
not specified, 9735 is used.

This option can be used multiple times to add more addresses, and
its use disables autolisten.  If necessary, and 'always-use-proxy'
is not specified, a DNS lookup may be done to resolve 'IPADDRESS'.
```

announce-addr=*IPADDRESS[:PORT]|TORADDRESS.onion[:PORT]* Set an IP (v4 or v6) address or Tor address to announce; a Tor address is distinguished by ending in *.onion*. *PORT* defaults to 9735.

```
Empty or wildcard IPv4 and IPv6 addresses don't make sense here.
Also, unlike the 'addr' option, there is no checking that your
announced addresses are public (e.g. not localhost).

This option can be used multiple times to add more addresses, and
its use disables autolisten.  The spec says you can't announce
more that one address of the same type (eg. two IPv4 or two IPv6
addresses) so `lightningd` will refuse if you specify more than one.

If necessary, and 'always-use-proxy' is not specified, a DNS
lookup may be done to resolve 'IPADDRESS'.
```

offline Do not bind to any ports, and do not try to reconnect to any peers. This can be useful for maintenance and forensics, so is usually specified on the command line. Overrides all *addr* and *bind-addr* options.

autolisten=*BOOL* By default, we bind (and maybe announce) on IPv4 and IPv6 interfaces if no *addr*, *bind-addr* or *announce-addr* options are specified. Setting this to *false* disables that.

proxy=*IPADDRESS[:PORT]* Set a socks proxy to use to connect to Tor nodes (or for all connections if **always-use-**

proxy is set).

always-use-proxy=*BOOL* Always use the **proxy**, even to connect to normal IP addresses (you can still connect to Unix domain sockets manually). This also disables all DNS lookups, to avoid leaking information.

disable-dns Disable the DNS bootstrapping mechanism to find a node by its node ID.

enable-autotor-v2-mode Try to get a v2 onion address from the Tor service call, default is v3.

tor-service-password=*PASSWORD* Set a Tor control password, which may be needed for *autotor:* to authenticate to the Tor control port.

8.4.6 Lightning Plugins

lightningd(8) supports plugins, which offer additional configuration options and JSON-RPC methods, depending on the plugin. Some are supplied by default (usually located in **libexec/c-lightning/plugins/**). If a **plugins** directory exists under *lightning-dir* that is searched for plugins along with any immediate subdirectories). You can specify additional paths too:

plugin=*PATH* Specify a plugin to run as part of c-lightning. This can be specified multiple times to add multiple plugins.

plugin-dir=*DIRECTORY* Specify a directory to look for plugins; all executable files not containing punctuation (other than ., - or *_) in 'DIRECTORY* are loaded. *DIRECTORY* must exist; this can be specified multiple times to add multiple directories.

clear-plugins This option clears all *plugin* and *plugin-dir* options preceeding it, including the default built-in plugin directory. You can still add *plugin-dir* and *plugin* options following this and they will have the normal effect.

disable-plugin=*PLUGIN* If *PLUGIN* contains a /, plugins with the same path as *PLUGIN* are disabled. Otherwise, any plugin with that base name is disabled, whatever directory it is in.

8.5 BUGS

You should report bugs on our github issues page, and maybe submit a fix to gain our eternal gratitude!

8.6 AUTHOR

Rusty Russell <rusty@rustcorp.com.au> wrote this man page, and much of the configuration language, but many others did the hard work of actually implementing these options.

8.7 SEE ALSO

lightning-listconfigs(7) lightning-setchannelfee(7) lightningd(8)

8.8 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

8.9 COPYING

Note: the modules in the ccan/ directory have their own licenses, but the rest of the code is covered by the BSD-style MIT license.

CHAPTER 9

lightningd – Daemon for running a Lightning Network node

9.1 SYNOPSIS

```
lightningd [--conf=<config-file>] [OPTIONS]...
```

9.2 DESCRIPTION

lightningd starts the C-Lightning daemon, which implements a standards-compliant Lightning Network node.

9.3 CONFIGURATION OPTIONS

–conf=*FILE* Specify configuration file. If not an absolute path, will be relative from the lightning-dir location. Defaults to *config*.

–lightning-dir=*DIR* Set the directory for the C-Lightning daemon. Defaults to *$HOME/.lightning*.

9.4 MORE OPTIONS

Command line options are mirrored as configuration options in the configuration file, so *foo* in the configuration file simply becomes **–foo** on the command line, and **foo=bar** becomes **–foo=bar**.

See lightningd-config(5) for a comprehensive list of all available options.

9.5 LOGGING AND COMMANDING C-LIGHTNING

By default, C-Lightning will log to the standard output. To log to a specific file, use **–log-file**=*PATH*. Sending SIGHUP will cause C-Lightning to reopen this file, for example to do log rotation.

C-Lightning will set up a Unix domain socket for receiving commands. By default this will be the file **lightning-rpc** in your specified **lightning-dir**. You can use lightning-cli(1) to send commands to C-Lightning once **lightningd** has started; you need to match the **–lightning-dir** and **–rpc-file** options between them.

Commands for C-Lightning are described in various manpages in section 7, with the common prefix **lightning-**.

9.6 QUICK START

First, decide on and create a directory for *lightning-dir*, or just use the default *$HOME/.lightning*. Then create a *config* file in this directory containing your configuration.

Your other main preparation would be to set up a mainnet Bitcoin fullnode, i.e. run a bitcoind(1) instance. The rest of this quick start guide will assume you are reckless and want to spend real funds on Lightning. Indicate *network=bitcoin* in your *config* file explicitly.

C-Lightning needs to communicate with the Bitcoin Core RPC. You can set this up using *bitcoin-datadir*, *bitcoin-rpcconnect*, *bitcoin-rpcport*, *bitcoin-rpcuser*, and *bitcoin-rpcpassword* options in your *config* file.

Finally, just to keep yourself sane, decide on a log file name and indicate it using *log-file=lightningd.log* in your *config* file. You might be interested in viewing it periodically as you follow along on this guide.

Once the **bitcoind** instance is running, start lightningd(8):

```
$ lightningd --lightning-dir=$HOME/.lightning --daemon
```

This starts **lightningd** in the background due to the *–daemon* option.

Check if things are working:

```
$ lightning-cli --lightning-dir=%HOME/.lightning help
$ lightning-cli --lightning-dir=%HOME/.lightning getinfo
```

The **getinfo** command in particular will return a *blockheight* field, which indicates the block height to which **lightningd** has been synchronized to (this is separate from the block height that your **bitcoind** has been synchronized to, and will always lag behind **bitcoind**). You will have to wait until the *blockheight* has reached the actual blockheight of the Bitcoin network.

Before you can get funds offchain, you need to have some funds onchain owned by **lightningd** (which has a separate wallet from the **bitcoind** it connects to). Get an address for **lightningd** via lightning-newaddr(7) command as below (*–lightning-dir* option has been elided, specify it if you selected your own *lightning-dir*):

```
$ lightning-cli newaddr
```

This will provide a native SegWit bech32 address. In case all your money is in services that do not support native SegWit and have to use P2SH-wrapped addresses, instead use:

```
$ lightning-cli newaddr p2sh-segwit
```

Transfer a small amount of onchain funds to the given address. Check the status of all your funds (onchain and on-Lightning) via lightning-listfunds(7):

```
$ lightning-cli listfunds
```

Now you need to look for an arbitrary Lightning node to connect to, which you can do by using dig(1) and querying *lseed.bitcoinstats.com*:

```
$ dig lseed.bitcoinstats.com A
```

This will give 25 IPv4 addresses, you can select any one of those. You will also need to learn the corresponding public key, which you can determine by searching the IP addrss on https://1ml.com/ . The public key is a long hex string, like so: *024772ee4fa461febcef09d5869e1238f932861f57be7a6633048514e3f56644a1*. (this example public key is not used as of this writing)

After determining a public key, use lightning-connect(7) to connect to that public key:

```
$ lightning-cli connect $PUBLICKEY
```

Then open a channel to that node using lightning-fundchannel(7):

```
$ lightning-cli fundchannel $PUBLICKEY $SATOSHI
```

This will require that the funding transaction be confirmed before you can send funds over Lightning. To track this, use lightning-listpeers(7) and look at the *state* of the channel:

```
$ lightning-cli listpeers $PUBLICKEY
```

The channel will initially start with a *state* of *CHANNELD_AWAITING_LOCKIN*. You need to wait for the channel *state* to become *CHANNELD_NORMAL*, meaning the funding transaction has been confirmed deeply.

Once the channel *state* is *CHANNELD_NORMAL*, you can start paying merchants over Lightning. Acquire a Lightning invoice from your favorite merchant, and use lightning-pay(7) to pay it:

```
$ lightning-cli pay $INVOICE
```

9.7 BUGS

You should report bugs on our github issues page, and maybe submit a fix to gain our eternal gratitude!

9.8 AUTHOR

ZmnSCPxj <ZmnSCPxj@protonmail.com> wrote the initial version of this man page, but many others did the hard work of actually implementing a standards-compliant Lightning Network node implementation.

9.9 SEE ALSO

lightning-listconfigs(7), lightning-config(5), lightning-cli(1), lightning-newaddr(7), lightning-listfunds(7), lightning-connect(7), lightning-fundchannel(7), lightning-listpeers(7), lightning-pay(7)

9.10 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

9.11 COPYING

Note: the modules in the ccan/ directory have their own licenses, but the rest of the code is covered by the BSD-style MIT license.

CHAPTER 10

lightning-autocleaninvoice – Set up auto-delete of expired invoice

10.1 SYNOPSIS

autocleaninvoice [*cycle_seconds*] [*expired_by*]

10.2 DESCRIPTION

The **autocleaninvoice** RPC command sets up automatic cleaning of expired invoices.

Autoclean will be done every *cycle_seconds* seconds. Setting *cycle_seconds* to 0 disables autoclean. If not specified, this defaults to 3600 (one hour).

Every autoclean cycle, expired invoices, which have already been expired for at least *expired_by* seconds, will be deleted. If *expired_by* is not specified, this defaults to 86400 (one day).

On startup of the daemon, no autoclean is set up.

10.3 RETURN VALUE

On success, an empty object is returned.

10.4 AUTHOR

ZmnSCPxj <ZmnSCPxj@protonmail.com> is mainly responsible.

10.5 SEE ALSO

lightning-delexpiredinvoice(7), lightning-delinvoice(7)

10.6 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

CHAPTER 11

lightning-check – Command for verifying parameters

11.1 SYNOPSIS

check *command_to_check* [*parameters*]

11.2 DESCRIPTION

The **check** RPC command verifies another command's parameters without running it.

The *command_to_check* is the name of the relevant command.

parameters is the command's parameters.

This does not guarantee successful execution of the command in all cases. For example, a call to lightning-getroute(7) may still fail to find a route even if checking the parameters succeeds.

11.3 RETURN VALUE

On success, the *command_to_check* is returned. On failure, the relevant RPC error is returned.

11.4 AUTHOR

Mark Beckwith <wythe@intrig.com> and Rusty Russell <rusty@rustcorp.com.au> are mainly responsible.

11.5 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

CHAPTER 12

lightning-checkmessage – Command to check a signature is from a node

12.1 SYNOPSIS

checkmessage *message zbase* [*pubkey*]

12.2 DESCRIPTION

The **checkmessage** RPC command is the counterpart to **signmessage**: given a node id (*pubkey*), signature (*zbase*) and a *message*, it verifies that the signature was generated by that node for that message (more technically: by someone who knows that node's secret).

As a special case, if *pubkey* is not specified, we will try every known node key (as per *listnodes*), and verification succeeds if it matches for any one of them. Note: this is implemented far more efficiently than trying each one, so performance is not a concern.

12.3 RETURN VALUE

On correct usage, an object with attribute *verified* will be returned.

If *verified* is true, the signature was generated by the returned *pubkey* for that given message. *pubkey* is the one specified as input, or if none was specified, the known node which must have produced this signature.

If *verified* is false, the signature is meaningless. *pubkey* may also be returned, which is they *pubkey* (if any) for which this signature would be valid. This is usually not useful.

12.4 AUTHOR

Rusty Russell <rusty@rustcorp.com.au> is mainly responsible.

12.5 SEE ALSO

lightning-signmessage(7)

12.6 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

CHAPTER 13

lightning-cli – Control lightning daemon

13.1 SYNOPSIS

lightning-cli [*OPTIONS*] *command*...

13.2 DESCRIPTION

lightning-cli sends commands to the lightning daemon.

13.3 OPTIONS

–lightning-dir=*DIR* Set the directory for the lightning daemon we're talking to; defaults to *$HOME/.lightning*.

–conf=*PATH* Sets configuration file (default: **lightning-dir**/*config*).

–network=*network* **–mainnet –testnet –signet** Sets network explicitly.

–rpc-file=*FILE* Named pipe to use to talk to lightning daemon: default is *lightning-rpc* in the lightning directory.

–keywords/-k Use format *key*=*value* for parameters in any order

–order/-o Follow strictly the order of parameters for the command

–json/-J Return result in JSON format (default unless *help* command)

–raw/-R Return raw JSON directly as lightningd replies

–human-readable/-H Return result in human-readable output (default for *help* command)

–help/-h Print summary of options to standard output and exit.

–version/-V Print version number to standard output and exit.

allow-deprecated-apis=*BOOL* Enable deprecated options. It defaults to *true*, but you should set it to *false* when testing to ensure that an upgrade won't break your configuration.

13.4 COMMANDS

lightning-cli simply uses the JSON RPC interface to talk to *lightningd*, and prints the results. Thus the commands available depend entirely on the lightning daemon itself.

13.5 ARGUMENTS

Arguments may be provided positionally or using *key*=*value* after the command name, based on either **-o** or **-k** option. Arguments may be integer numbers (composed entirely of digits), floating-point numbers (has a radix point but otherwise composed of digits), *true*, *false*, or *null*. Other arguments are treated as strings.

Some commands have optional arguments. You may use *null* to skip optional arguments to provide later arguments.

13.6 EXAMPLES

Example 1. List commands

lightning-cli help

13.7 BUGS

This manpage documents how it should work, not how it does work. The pretty printing of results isn't pretty.

13.8 AUTHOR

Rusty Russell <rusty@rustcorp.com.au> is mainly to blame.

13.9 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

13.10 COPYING

Note: the modules in the ccan/ directory have their own licenses, but the rest of the code is covered by the BSD-style MIT license.

CHAPTER 14

lightning-close – Command for closing channels with direct peers

14.1 SYNOPSIS

close *id* [*unilateraltimeout*] [*destination*]

14.2 DESCRIPTION

The **close** RPC command attempts to close the channel cooperatively with the peer, or unilaterally after *unilateraltimeout*, and the to-local output will be sent to the address specified in *destination*.

If the given *id* is a peer ID (66 hex digits as a string), then it applies to the active channel of the direct peer corresponding to the given peer ID. If the given *id* is a channel ID (64 hex digits as a string, or the short channel ID *blockheight:txindex:outindex* form), then it applies to that channel.

If *unilateraltimeout* is not zero, the **close** command will unilaterally close the channel when that number of seconds is reached. If *unilateraltimeout* is zero, then the **close** command will wait indefinitely until the peer is online and can negotiate a mutual close. The default is 2 days (172800 seconds).

The *destination* can be of any Bitcoin accepted type, including bech32. If it isn't specified, the default is a c-lightning wallet address.

The peer needs to be live and connected in order to negotiate a mutual close. The default of unilaterally closing after 48 hours is usually a reasonable indication that you can no longer contact the peer.

14.3 NOTES

Prior to 0.7.2, **close** took two parameters: *force* and *timeout*. *timeout* was the number of seconds before *force* took effect (default, 30), and *force* determined whether the result was a unilateral close or an RPC error (default). Even after the timeout, the channel would be closed if the peer reconnected.

14.4 RETURN VALUE

On success, an object with fields *tx* and *txid* containing the closing transaction are returned. It will also have a field *type* which is either the JSON string *mutual* or the JSON string *unilateral*. A *mutual* close means that we could negotiate a close with the peer, while a *unilateral* close means that the *force* flag was set and we had to close the channel without waiting for the counterparty.

A unilateral close may still occur at any time if the peer did not behave correctly during the close negotiation.

Unilateral closes will return your funds after a delay. The delay will vary based on the peer *to_self_delay* setting, not your own setting.

14.5 AUTHOR

ZmnSCPxj <ZmnSCPxj@protonmail.com> is mainly responsible.

14.6 SEE ALSO

lightning-disconnect(7), lightning-fundchannel(7)

14.7 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

CHAPTER 15

lightning-connect – Command for connecting to another lightning node

15.1 SYNOPSIS

connect *id* [*host port*]

15.2 DESCRIPTION

The **connect** RPC command establishes a new connection with another node in the Lightning Network.

id represents the target node's public key. As a convenience, *id* may be of the form *id@host* or *id@host:port*. In this case, the *host* and *port* parameters must be omitted.

host is the peer's hostname or IP address.

If not specified, the *port* defaults to 9735.

If *host* is not specified, the connection will be attempted to an IP belonging to *id* obtained through gossip with other already connected peers.

If *host* begins with a */* it is interpreted as a local path, and the connection will be made to that local socket (see **bind-addr** in lightningd-config(5)).

Connecting to a node is just the first step in opening a channel with another node. Once the peer is connected a channel can be opened with lightning-fundchannel(7).

15.3 RETURN VALUE

On success the peer *id* is returned.

15.4 ERRORS

On failure, one of the following errors will be returned:

```
{ "code" : 400, "message" : "Unable to connect, no address known for peer" }
```

If some addresses are known but connecting to all of them failed, the message will contain details about the failures:

```
{ "code" : 401, "message" : "..." }
```

If the given parameters are wrong:

```
{ "code" : -32602, "message" : "..." }
```

15.5 AUTHOR

Rusty Russell <rusty@rustcorp.com.au> is mainly responsible. Felix <fixone@gmail.com> is the original author of this manpage.

15.6 SEE ALSO

lightning-fundchannel(7), lightning-listpeers(7), lightning-listchannels(7), lightning-disconnect(7)

15.7 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

CHAPTER 16

lightning-createonion – Low-level command to create a custom onion

16.1 SYNOPSIS

createonion *hops assocdata* [*session_key*]

16.2 DESCRIPTION

The **createonion** RPC command allows the caller to create a custom onion with custom payloads at each hop in the route. A custom onion can be used to implement protocol extensions that are not supported by c-lightning directly.

The *hops* parameter is a JSON list of dicts, each specifying a node and the payload destined for that node. The following is an example of a 3 hop onion:

```
[
        {
                "pubkey":
↪"022d223620a359a47ff7f7ac447c85c46c923da53389221a0054c11c1e3ca31d59",
                "payload":
↪"000000670000010001000000000000003e90000007b000000000000000000000000000000000000000000000000000
↪"
        }, {
                "pubkey":
↪"035d2b1192dfba134e10e540875d366ebc8bc353d5aa766b80c090b39c3a5d885d",
                "payload":
↪"000000670000030001000000000000003e8000000750000000000000000000000000000000000000000000000000000
↪"
        }, {
                "style": "legacy",
                "pubkey":
↪"0382ce59ebf18be7d84677c2e35f23294b9992ceca95491fcf8a56c6cb2d9de199",
                "payload":
↪"000000670000030001000000000000003e8000000750000000000000000000000000000000000000000000000000000
↪"
```

(continues on next page)

(continued from previous page)

```
        }
]
```

The *hops* parameter is very similar to the result from `getroute` however it needs to be modified slightly. The following is the `getroute` response from which the above *hops* parameter was generated:

```
[
        {
                "id":
↪"022d223620a359a47ff7f7ac447c85c46c923da53389221a0054c11c1e3ca31d59",
                "channel": "103x2x1",
                "direction": 1,
                "msatoshi": 1002,
                "amount_msat": "1002msat",
                "delay": 21,
                "style": "legacy"
        }, {
                "id":
↪"035d2b1192dfba134e10e540875d366ebc8bc353d5aa766b80c090b39c3a5d885d",
                "channel": "103x1x1",
                "direction": 0,
                "msatoshi": 1001,
                "amount_msat": "1001msat",
                "delay": 15,
                "style": "legacy"
        }, {
                "id":
↪"0382ce59ebf18be7d84677c2e35f23294b9992ceca95491fcf8a56c6cb2d9de199",
                "channel": "103x3x1",
                "direction": 0,
                "msatoshi": 1000,
                "amount_msat": "1000msat",
                "delay": 9,
                "style": "legacy"
        }
]
```

- Notice that the payload in the *hops* parameter is the hex-encoded version of the parameters in the `getroute` response.
- The payloads are shifted left by one, i.e., payload 0 in `createonion` corresponds to payload 1 from `getroute`.
- The final payload is a copy of the last payload sans `channel`

These rules are directly derived from the onion construction. Please refer BOLT 04 for details and rationale.

The *assocdata* parameter specifies the associated data that the onion should commit to. If the onion is to be used to send a payment later it MUST match the `payment_hash` of the payment in order to be valid.

The optional *session_key* parameter can be used to specify a secret that is used to generate the shared secrets used to encrypt the onion for each hop. It should only be used for testing or if a specific shared secret is important. If not specified it will be securely generated internally, and the shared secrets will be returned.

16.3 RETURN VALUE

On success, an object containing the onion and the shared secrets will be returned. Otherwise an error will be reported. The following example is the result of calling *createonion* with the above hops parameter:

```
{
        "onion": "0003f3f80d2142b953319336d2fe4097[......
↪]6af33fcf4fb113bce01f56dd62248a9e5fcbbfba35c",
        "shared_secrets": [
                "88ce98c73e4d9293ab1797b0a913fe9bca0213a566252047d01b8af6da871f3e",
                "4474d296810e57bd460ef8b83d2e7d288321f8a99ff7686f87384699747bcfc4",
                "2a862e4123e01799a732be487fbce297f7dc7cc1467e410f18369cfee476adc2"
        ]
}
```

The `onion` corresponds to 1366 hex-encoded bytes. Each shared secret consists of 32 hex-encoded bytes. Both arguments can be passed on to **sendonion**.

16.4 AUTHOR

Christian Decker <decker.christian@gmail.com> is mainly responsible.

16.5 SEE ALSO

lightning-sendonion(7), lightning-getroute(7)

16.6 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

CHAPTER 17

lightning-decodepay – Command for decoding a bolt11 string (low-level)

17.1 SYNOPSIS

decodepay *bolt11* [*description*]

17.2 DESCRIPTION

The **decodepay** RPC command checks and parses a *bolt11* string as specified by the BOLT 11 specification.

17.3 RETURN VALUE

On success, an object is returned with the following fields, as specified by BOLT11:

- *currency*: the BIP173 name for the currency.
- *timestamp*: the UNIX-style timestamp of the invoice.
- *expiry*: the number of seconds this is valid after *timestamp*.
- *payee*: the public key of the recipient.
- *payment_hash*: the payment hash of the request.
- *signature*: the DER-encoded signature.
- *description*: the description of the purpose of the purchase (see below)

The following fields are optional:

- *msatoshi*: the number of millisatoshi requested (if any).
- *amount_msat*: the same as above, with *msat* appended (if any).

- *fallbacks*: array of fallback address object containing a *hex* string, and both *type* and *addr* if it is recognized as one of *P2PKH*, *P2SH*, *P2WPKH*, or *P2WSH*.
- *routes*: an array of routes. Each route is an arrays of objects, each containing *pubkey*, *short_channel_id*, *fee_base_msat*, *fee_proportional_millionths* and *cltv_expiry_delta*.
- *extra*: an array of objects representing unknown fields, each with one-character *tag* and a *data* bech32 string.

Technically, the *description* field is optional if a *description_hash* field is given, but in this case **decodepay** will only succeed if the optional *description* field is passed and matches the *description_hash*. In practice, these are currently unused.

17.4 AUTHOR

Rusty Russell <rusty@rustcorp.com.au> is mainly responsible.

17.5 SEE ALSO

lightning-pay(7), lightning-getroute(7), lightning-sendpay(7).

BOLT #11.

17.6 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

CHAPTER 18

lightning-delexpiredinvoice – Command for removing expired invoices

18.1 SYNOPSIS

delexpiredinvoice [*maxexpirytime*]

18.2 DESCRIPTION

The **delexpiredinvoice** RPC command removes all invoices that have expired on or before the given *maxexpirytime*.

If *maxexpirytime* is not specified then all expired invoices are deleted.

18.3 RETURN VALUE

On success, an empty object is returned.

18.4 AUTHOR

ZmnSCPxj <ZmnSCPxj@protonmail.com> is mainly responsible.

18.5 SEE ALSO

lightning-delinvoice(7), lightning-autocleaninvoice(7)

18.6 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

CHAPTER 19

lightning-delinvoice – Command for removing an invoice

19.1 SYNOPSIS

delinvoice *label status*

19.2 DESCRIPTION

The **delinvoice** RPC command removes an invoice with *status* as given in **listinvoices**.

The caller should be particularly aware of the error case caused by the *status* changing just before this command is invoked!

19.3 RETURN VALUE

On success, an invoice description will be returned as per lightning-listinvoice(7).

19.4 AUTHOR

Rusty Russell <rusty@rustcorp.com.au> is mainly responsible.

19.5 SEE ALSO

lightning-listinvoice(7), lightning-waitinvoice(7), lightning-invoice(7), lightning-delexpiredinvoice(7), lightning-autocleaninvoice(7)

19.6 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

CHAPTER 20

lightning-disconnect – Command for disconnecting from another lightning node

20.1 SYNOPSIS

disconnect *id* [*force*]

20.2 DESCRIPTION

The disconnect RPC command closes an existing connection to a peer, identified by *id*, in the Lightning Network, as long as it doesn't have an active channel. If *force* is set then it will disconnect even with an active channel.

The *id* can be discovered in the output of the listpeers command, which returns a set of peers:

```
{
     "peers": [
          {
               "id": "0563aea81...",
               "connected": true,
               ...
          }
     ]
}
```

Passing the *id* attribute of a peer to *disconnect* will terminate the connection.

20.3 RETURN VALUE

On success, an empty object is returned.

On error the returned object will contain `code` and `message` properties, with `code` being one of the following:

- -32602: If the given parameters are wrong.
- -1: Catchall nonspecific error.

20.4 AUTHOR

Michael Hawkins <michael.hawkins@protonmail.com>.

20.5 SEE ALSO

lightning-connect(1), lightning-listpeers(1)

20.6 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

CHAPTER 21

lightning-fundchannel – Command for establishing a lightning channel

21.1 SYNOPSIS

fundchannel *id amount* [*feerate announce*] [*minconf*] [*utxos*] [*push_msat*]

21.2 DESCRIPTION

The **fundchannel** RPC command opens a payment channel with a peer by committing a funding transaction to the blockchain as defined in BOLT #2. **fundchannel** by itself does not attempt to open a connection. A connection must first be established using **connect**. Once the transaction is confirmed, normal channel operations may begin. Readiness is indicated by **listpeers** reporting a *state* of CHANNELD_NORMAL for the channel.

id is the peer id obtained from **connect**.

amount is the amount in satoshis taken from the internal wallet to fund the channel. The string *all* can be used to specify all available funds (or 16777215 satoshi if more is available). Otherwise, it is in satoshi precision; it can be a whole number, a whole number ending in *sat*, a whole number ending in *000msat*, or a number with 1 to 8 decimal places ending in *btc*. The value cannot be less than the dust limit, currently set to 546, nor more than 16777215 satoshi.

feerate is an optional feerate used for the opening transaction and as initial feerate for commitment and HTLC transactions. It can be one of the strings *urgent* (aim for next block), *normal* (next 4 blocks or so) or *slow* (next 100 blocks or so) to use lightningd's internal estimates: *normal* is the default.

Otherwise, *feerate* is a number, with an optional suffix: *perkw* means the number is interpreted as satoshi-per-kilosipa (weight), and *perkb* means it is interpreted bitcoind-style as satoshi-per-kilobyte. Omitting the suffix is equivalent to *perkb*.

announce is an optional flag that triggers whether to announce this channel or not. Defaults to `true`. An unannounced channel is considered private.

minconf specifies the minimum number of confirmations that used outputs should have. Default is 1.

utxos specifies the utxos to be used to fund the channel, as an array of "txid:vout".

push_msat is the amount of millisatoshis to push to the channel peer at open. Note that this is a gift to the peer – these satoshis are added to the initial balance of the peer at channel start and are largely unrecoverable once pushed.

21.3 RETURN VALUE

On success, the *tx* and *txid* of the transaction is returned, as well as the *channel_id* of the newly created channel. On failure, an error is reported and the channel is not funded.

The following error codes may occur:

- -1: Catchall nonspecific error.
- 300: The maximum allowed funding amount is exceeded.
- 301: There are not enough funds in the internal wallet (including fees) to create the transaction.
- 302: The output amount is too small, and would be considered dust.
- 303: Broadcasting of the funding transaction failed, the internal call to bitcoin-cli returned with an error.

Failure may also occur if **lightningd** and the peer cannot agree on channel parameters (funding limits, channel reserves, fees, etc.).

21.4 SEE ALSO

lightning-connect(7), lightning-listfunds(), lightning-listpeers(7)

21.5 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

CHAPTER 22

lightning-fundchannel_cancel – Command for completing channel establishment

22.1 SYNOPSIS

fundchannel_cancel *id*

22.2 DESCRIPTION

`fundchannel_cancel` is a lower level RPC command. It allows channel funder to cancel a channel before funding broadcast with a connected peer.

id is the node id of the remote peer with which to cancel.

Note that the funding transaction MUST NOT be broadcast before `fundchannel_cancel`. Broadcasting transaction before `fundchannel_cancel` WILL lead to unrecoverable loss of funds.

If `fundchannel_cancel` is called after `fundchannel_complete`, the remote peer may disconnect when command succeeds. In this case, user need to connect to remote peer again before opening channel.

22.3 RETURN VALUE

On success, returns confirmation that the channel establishment has been canceled.

On error the returned object will contain `code` and `message` properties, with `code` being one of the following:

- -32602: If the given parameters are wrong.
- -1: Catchall nonspecific error.
- 306: Unknown peer id.

22.4 AUTHOR

Lisa Neigut <niftynei@gmail.com> is mainly responsible.

22.5 SEE ALSO

lightning-connect(7), lightning-fundchannel(7), lightning-fundchannel_start(7), lightning-fundchannel_complete(7)

22.6 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

CHAPTER 23

lightning-fundchannel_complete – Command for completing channel establishment

23.1 SYNOPSIS

fundchannel_complete *id txid txout*

23.2 DESCRIPTION

`fundchannel_complete` is a lower level RPC command. It allows a user to complete an initiated channel establishment with a connected peer.

id is the node id of the remote peer.

txid is the hex string of the funding transaction id.

txout is the integer outpoint of the funding output for this channel.

Note that the funding transaction MUST NOT be broadcast until after channel establishment has been successfully completed, as the commitment transactions for this channel are not secured until this command successfully completes. Broadcasting transaction before can lead to unrecoverable loss of funds.

23.3 RETURN VALUE

On success, returns a confirmation that *commitments_secured* and the derived *channel_id*.

On error the returned object will contain `code` and `message` properties, with `code` being one of the following:

- -32602: If the given parameters are wrong.
- -1: Catchall nonspecific error.
- 305: Peer is not connected.

- 306: Unknown peer id.

23.4 AUTHOR

Lisa Neigut <niftynei@gmail.com> is mainly responsible.

23.5 SEE ALSO

lightning-connect(7), lightning-fundchannel(7), lightning-fundchannel_start(7), lightning-fundchannel_cancel(7)

23.6 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

CHAPTER 24

lightning-fundchannel_start – Command for initiating channel establishment for a lightning channel

24.1 SYNOPSIS

fundchannel_start *id amount* [*feerate announce close_to push_msat*]

24.2 DESCRIPTION

`fundchannel_start` is a lower level RPC command. It allows a user to initiate channel establishment with a connected peer.

id is the node id of the remote peer.

amount is the satoshi value that the channel will be funded at. This value MUST be accurate, otherwise the negotiated commitment transactions will not encompass the correct channel value.

feerate is an optional field. Sets the feerate for subsequent commitment transactions: see **fundchannel**.

announce whether or not to announce this channel.

close_to is a Bitcoin address to which the channel funds should be sent to on close. Only valid if both peers have negotiated `option_upfront_shutdown_script`. Returns `close_to` set to closing script iff is negotiated.

push_msat is the amount of millisatoshis to push to the channel peer at open. Note that this is a gift to the peer – these satoshis are added to the initial balance of the peer at channel start and are largely unrecoverable once pushed.

Note that the funding transaction MUST NOT be broadcast until after channel establishment has been successfully completed by running `fundchannel_complete`, as the commitment transactions for this channel are not secured until the complete command succeeds. Broadcasting transaction before that can lead to unrecoverable loss of funds.

24.3 RETURN VALUE

On success, returns the *funding_address* and the *scriptpubkey* for the channel funding output. If a `close_to` address was provided, will close to this address iff the `close_to` address is returned in the response. Otherwise, the peer does not support `option_upfront_shutdownscript`.

On error the returned object will contain `code` and `message` properties, with `code` being one of the following:

- -32602: If the given parameters are wrong.
- -1: Catchall nonspecific error.
- 300: The amount exceeded the maximum configured funding amount.
- 301: The provided `push_msat` is greater than the provided `amount`.
- 304: Still syncing with bitcoin network
- 305: Peer is not connected.
- 306: Unknown peer id.

24.4 AUTHOR

Lisa Neigut <niftynei@gmail.com> is mainly responsible.

24.5 SEE ALSO

lightning-connect(7), lightning-fundchannel(7), lightning-fundchannel_complete(7), lightning-fundchannel_cancel(7)

24.6 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

CHAPTER 25

lightning-getroute – Command for routing a payment (low-level)

25.1 SYNOPSIS

getroute *id msatoshi riskfactor* [*cltv*] [*fromid*] [*fuzzpercent*] [*exclude*] [*maxhops*]

25.2 DESCRIPTION

The **getroute** RPC command attempts to find the best route for the payment of *msatoshi* to lightning node *id*, such that the payment will arrive at *id* with *cltv*-blocks to spare (default 9).

msatoshi is in millisatoshi precision; it can be a whole number, or a whole number ending in *msat* or *sat*, or a number with three decimal places ending in *sat*, or a number with 1 to 11 decimal places ending in *btc*.

There are two considerations for how good a route is: how low the fees are, and how long your payment will get stuck in a delayed output if a node goes down during the process. The *riskfactor* floating-point field controls this tradeoff; it is the annual cost of your funds being stuck (as a percentage).

For example, if you thought the convenience of keeping your funds liquid (not stuck) was worth 20% per annum interest, *riskfactor* would be 20.

If you didn't care about risk, *riskfactor* would be zero.

fromid is the node to start the route from: default is this node.

The *fuzzpercent* is a positive floating-point number, representing a percentage of the actual fee. The *fuzzpercent* is used to distort computed fees along each channel, to provide some randomization to the route generated. 0.0 means the exact fee of that channel is used, while 100.0 means the fee used might be from 0 to twice the actual fee. The default is 5.0, or up to 5% fee distortion.

exclude is a JSON array of short-channel-id/direction (e.g. ["564334x877x1/0", "564195x1292x0/1"]) or node-id which should be excluded from consideration for routing. The default is not to exclude any channels or nodes. Note if the source or destination is excluded, the command result is undefined.

maxhops is the maximum number of channels to return; default is 20.

25.3 RISKFACTOR EFFECT ON ROUTING

The risk factor is treated as if it were an additional fee on the route, for the purposes of comparing routes.

The formula used is the following approximation:

```
risk-fee = amount x blocks-timeout x per-block-cost
```

We are given a *riskfactor* expressed as a percentage. There are 52596 blocks per year, thus *per-block-cost* is *riskfactor* divided by 5,259,600.

The final result is:

```
risk-fee = amount x blocks-timeout x riskfactor / 5259600
```

Here are the risk fees in millisatoshis, using various parameters. I assume a channel charges the default of 1000 millisatoshis plus 1 part-per-million. Common to_self_delay values on the network at 14 and 144 blocks.

25.4 RECOMMENDED RISKFACTOR VALUES

The default *fuzz* factor is 5%, so as you can see from the table above, that tends to overwhelm the effect of *riskfactor* less than about 5.

1 is a conservative value for a stable lightning network with very few failures.

1000 is an aggressive value for trying to minimize timeouts at all costs.

The default for lightning-pay(7) is 10, which starts to become a major factor for larger amounts, and is basically ignored for tiny ones.

25.5 RETURN VALUE

On success, a "route" array is returned. Each array element contains *id* (the node being routed through), *msatoshi* (the millisatoshis sent), *amount_msat* (the same, with *msat* appended), *delay* (the number of blocks to timeout at this node), and *style* (indicating the features which can be used for this hop).

The final *id* will be the destination *id* given in the input. The difference between the first *msatoshi* minus the *msatoshi* given in the input is the fee. The first *delay* is the very worst case timeout for the payment failure, in blocks.

25.6 AUTHOR

Rusty Russell <rusty@rustcorp.com.au> is mainly responsible.

25.7 SEE ALSO

lightning-pay(7), lightning-sendpay(7).

25.8 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

CHAPTER 26

lightning-invoice – Command for accepting payments

26.1 SYNOPSIS

invoice *msatoshi label description* [*expiry*] [*fallbacks*] [*preimage*] [*exposeprivatechannels*]

26.2 DESCRIPTION

The **invoice** RPC command creates the expectation of a payment of a given amount of milli-satoshi: it returns a unique token which another lightning daemon can use to pay this invoice. This token includes a *route hint* description of an incoming channel with capacity to pay the invoice, if any exists.

The *msatoshi* parameter can be the string "any", which creates an invoice that can be paid with any amount. Otherwise it is in millisatoshi precision; it can be a whole number, or a whole number ending in *msat* or *sat*, or a number with three decimal places ending in *sat*, or a number with 1 to 11 decimal places ending in *btc*.

The *label* must be a unique string or number (which is treated as a string, so "01" is different from "1"); it is never revealed to other nodes on the lightning network, but it can be used to query the status of this invoice.

The *description* is a short description of purpose of payment, e.g. *1 cup of coffee*. This value is encoded into the BOLT11 invoice and is viewable by any node you send this invoice to. It must be UTF-8, and cannot use *\u* JSON escape codes.

The *expiry* is optionally the time the invoice is valid for; without a suffix it is interpreted as seconds, otherwise suffixes *s*, *m*, *h*, *d*, *w* indicate seconds, minutes, hours, days and weeks respectively. If no value is provided the default of 604800 (1w) is used.

The *fallbacks* array is one or more fallback addresses to include in the invoice (in order from most-preferred to least): note that these arrays are not currently tracked to fulfill the invoice.

The *preimage* is a 64-digit hex string to be used as payment preimage for the created invoice. By default, if unspecified, lightningd will generate a secure pseudorandom preimage seeded from an appropriate entropy source on your system. **IMPORTANT**: if you specify the *preimage*, you are responsible, to ensure appropriate care for generating using a secure pseudorandom generator seeded with sufficient entropy, and keeping the preimage secret. This parameter is an

advanced feature intended for use with cutting-edge cryptographic protocols and should not be used unless explicitly needed.

If specified, *exposeprivatechannels* overrides the default route hint logic, which will use unpublished channels only if there are no published channels. If *true* unpublished channels are always considered as a route hint candidate; if *false*, never. If it is a short channel id (e.g. *1x1x3*) or array of short channel ids, only those specific channels will be considered candidates, even if they are public.

The route hint is selected from the set of incoming channels of which: peer's balance minus their reserves is at least *msatoshi*, state is normal, the peer is connected and not a dead end (i.e. has at least one other public channel). The selection uses some randomness to prevent probing, but favors channels that become more balanced after the payment.

26.3 RETURN VALUE

On success, a hash is returned as *payment_hash* to be given to the payer, and the *expiry_time* as a UNIX timestamp. It also returns a BOLT11 invoice as *bolt11* to be given to the payer.

On failure, an error is returned and no invoice is created. If the lightning process fails before responding, the caller should use lightning-listinvoice(7) to query whether this invoice was created or not.

The following error codes may occur:

- -1: Catchall nonspecific error.
- 900: An invoice with the given *label* already exists.
- 901: An invoice with the given *preimage* already exists.
- 902: None of the specified *exposeprivatechannels* were usable.

One of the following warnings may occur (on success):

- *warning_offline* if no channel with a currently connected peer has the incoming capacity to pay this invoice
- *warning_capacity* if there is no channel that has sufficient incoming capacity
- *warning_deadends* if there is no channel that is not a dead-end

26.4 AUTHOR

Rusty Russell <rusty@rustcorp.com.au> is mainly responsible.

26.5 SEE ALSO

lightning-listinvoice(7), lightning-delinvoice(7), lightning-getroute(7), lightning-sendpay(7).

26.6 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

CHAPTER 27

lightning-listchannels – Command to query active lightning channels in the entire network

27.1 SYNOPSIS

listchannels [*short_channel_id*] [*source*]

27.2 DESCRIPTION

The **listchannels** RPC command returns data on channels that are known to the node. Because channels may be bidirectional, up to 2 objects will be returned for each channel (one for each direction).

If *short_channel_id* is supplied, then only known channels with a matching *short_channel_id* are returned.

If *source* is supplied, then only channels leading from that node id are returned.

If neither is supplied, data on all lightning channels known to this node, are returned. These can be local channels or public channels broadcast on the gossip network.

27.3 RETURN VALUE

On success, an object with a "channels" key is returned containing a list of 0 or more objects.

Each object in the list contains the following data:

- *source* : The node providing entry to the channel, specifying the fees charged for using the channel in that direction.
- *destination* : The node providing the exit point for the channel.
- *short_channel_id* : The channel identifier.

- *public* : Boolean value, is publicly available. Non-local channels will only ever have this value set to true. Local channels are side-loaded by this node, rather than obtained through the gossip network, and so may have this value set to false.
- *satoshis* : Funds available in the channel.
- *amount_sat* : Same as above, but ending in *sat*.
- *message_flags* : Bitfield showing the presence of optional fields in the *channel_update* message (BOLT #7).
- *channel_flags* : Bitfields indicating the direction of the channel and signaling various options concerning the channel. (BOLT #7).
- *active* : Boolean value, is available for routing. This is linked to the channel flags data, where if the second bit is set, signals a channels temporary unavailability (due to loss of connectivity) OR permanent unavailability where the channel has been closed but not settlement on-chain.
- *last_update* : Unix timestamp (seconds) showing when the last channel_update message was received.
- *base_fee_millisatoshi* : The base fee (in millisatoshi) charged for the HTLC (BOLT #2).
- *fee_per_millionth* : The amount (in millionths of a satoshi) charged per transferred satoshi (BOLT #2).
- *delay* : The number of blocks delay required to wait for on-chain settlement when unilaterally closing the channel (BOLT #2).
- *htlc_minimum_msat* : The minimum payment which can be send through this channel.
- *htlc_maximum_msat* : The maximum payment which can be send through this channel.

If *short_channel_id* or *source* is supplied and no matching channels are found, a "channels" object with an empty list is returned.

On error the returned object will contain `code` and `message` properties, with `code` being one of the following:

- -32602: If the given parameters are wrong.

27.4 AUTHOR

Michael Hawkins <michael.hawkins@protonmail.com>.

27.5 SEE ALSO

lightning-fundchannel(7), lightning-listnodes(7)

27.6 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

Lightning RFC site

- BOLT #2: https://github.com/lightningnetwork/lightning-rfc/blob/master/02-peer-protocol.md
- BOLT #7: https://github.com/lightningnetwork/lightning-rfc/blob/master/07-routing-gossip.md

CHAPTER 28

lightning-listforwards – Command showing all htlcs and their information

28.1 SYNOPSIS

listforwards

28.2 DESCRIPTION

The **listforwards** RPC command displays all htlcs that have been attempted to be forwarded by the c-lightning node.

28.3 RETURN VALUE

On success one array will be returned: *forwards* with htlcs that have been processed

Each entry in *forwards* will include:

- *in_channel*: the short_channel_id of the channel that recieved the incoming htlc.
- *in_msatoshi*, *in_msat* - amount of msatoshis that are forwarded to this node.
- *status*: status can be either *offered* if the routing process is still ongoing, *settled* if the routing process is completed or *failed* if the routing process could not be completed.
- *received_time*: timestamp when incoming htlc was received.

The following additional fields are usually present, but will not be for some variants of status *local_failed* (if it failed before we determined these):

- *out_channel*: the short_channel_id of to which the outgoing htlc is supposed to be forwarded.
- *fee*, *fee_msat*: fee offered for forwarding the htlc in msatoshi.
- *out_msatoshi*, *out_msat* - amount of msatoshis to be forwarded.

The following fields may be offered, but for old forgotten HTLCs they will be omitted:

- *payment_hash* - the payment_hash belonging to the HTLC.

If the status is not 'offered', the following additional fields are present:

- *resolved_time* - timestamp when htlc was resolved (settled or failed).

28.4 AUTHOR

Rene Pickhardt <r.pickhardt@gmail.com> is mainly responsible.

28.5 SEE ALSO

lightning-getinfo(7)

28.6 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

CHAPTER 29

lightning-listfunds – Command showing all funds currently managed by the c-lightning node

29.1 SYNOPSIS

listfunds

29.2 DESCRIPTION

The **listfunds** RPC command displays all funds available, either in unspent outputs (UTXOs) in the internal wallet or funds locked in currently open channels.

29.3 RETURN VALUE

On success two arrays will be returned: *outputs* with funds currently locked onchain in UTXOs and *channels* with funds readily spendable in channels.

Each entry in *outputs* will include:

- *txid*
- *output* (the index of the output in the transaction)
- *value* (the output value in satoshis)
- *amount_msat* (the same as *value*, but in millisatoshi with *msat* appended)
- *address*
- *status* (whether *unconfirmed*, *confirmed*, or *spent*)

Each entry in *channels* will include:

- *peer_id* - the peer with which the channel is opened.

- *short_channel_id* - as per BOLT 7 (representing the block, transaction number and output index of the channel funding transaction).
- *channel_sat* - available satoshis on our node's end of the channel (values rounded down to satoshis as internal storage is in millisatoshi).
- *our_amount_msat* - same as above, but in millisatoshis with *msat* appended.
- *channel_total_sat* - total channel value in satoshi
- *amount_msat* - same as above, but in millisatoshis with *msat* appended.
- *funding_txid* - funding transaction id.
- *funding_output* - the index of the output in the funding transaction.
- *connected* - whether the channel peer is connected.
- *state* - the channel state, in particular *CHANNELD_NORMAL* means the channel can be used normally.

29.4 AUTHOR

Felix <fixone@gmail.com> is mainly responsible.

29.5 SEE ALSO

lightning-newaddr(7), lightning-fundchannel(7), lightning-withdraw(7)

29.6 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

CHAPTER 30

lightning-listinvoices – Command for querying invoice status

30.1 SYNOPSIS

listinvoices [*label*]

30.2 DESCRIPTION

The **listinvoices** RPC command gets the status of a specific invoice, if it exists, or the status of all invoices if given no argument.

30.3 RETURN VALUE

On success, an array *invoices* of objects is returned. Each object contains *label*, *payment_hash*, *status* (one of *unpaid*, *paid* or *expired*), *payment_preimage* (for paid invoices), and *expiry_time* (a UNIX timestamp). If the *msatoshi* argument to lightning-invoice(7) was not "any", there will be an *msatoshi* field as a number, and *amount_msat* as the same number ending in *msat*. If the invoice *status* is *paid*, there will be a *pay_index* field and an *msatoshi_received* field (which may be slightly greater than *msatoshi* as some overpaying is permitted to allow clients to obscure payment paths); there will also be an *amount_received_msat* field with the same number as *msatoshi_received* but ending in *msat*.

30.4 AUTHOR

Rusty Russell <rusty@rustcorp.com.au> is mainly responsible.

30.5 SEE ALSO

lightning-waitinvoice(7), lightning-delinvoice(7), lightning-invoice(7).

30.6 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

CHAPTER 31

lightning-listpays – Command for querying payment status

31.1 SYNOPSIS

listpays [bolt11]

31.2 DESCRIPTION

The **listpay** RPC command gets the status of all *pay* commands, or a single one if *bolt11* is specified.

31.3 RETURN VALUE

On success, an array of objects is returned. Each object contains:

bolt11 the *bolt11* argument given to *pay* (see below for exceptions).

status one of *complete*, *failed* or *pending*.

payment_preimage (if *status* is *complete*) proves payment was received.

label optional *label*, if provided to *pay*.

amount_sent_msat total amount sent, in "NNNmsat" format.

For old payments (pre-0.7) we didn't save the *bolt11* string, so in its place are three other fields:

payment_hash the hash of the *payment_preimage* which will prove payment.

destination the final destination of the payment.

amount_msat the amount the destination received, in "NNNmsat" format.

These three can all be extracted from *bolt11*, hence are obsolete.

31.4 AUTHOR

Rusty Russell <rusty@rustcorp.com.au> is mainly responsible.

31.5 SEE ALSO

lightning-pay(7), lightning-paystatus(7), lightning-listsendpays(7).

31.6 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

CHAPTER 32

lightning-listpeers – Command returning data on connected lightning nodes

32.1 SYNOPSIS

listpeers [*id*] [*level*]

32.2 DESCRIPTION

The **listpeers** RPC command returns data on nodes that are connected or are not connected but have open channels with this node.

Once a connection to another lightning node has been established, using the **connect** command, data on the node can be returned using **listpeers** and the *id* that was used with the **connect** command.

If no *id* is supplied, then data on all lightning nodes that are connected, or not connected but have open channels with this node, are returned.

Supplying *id* will filter the results to only return data on a node with a matching *id*, if one exists.

Supplying *level* will show log entries related to that peer at the given log level. Valid log levels are "io", "debug", "info", and "unusual".

If a channel is open with a node and the connection has been lost, then the node will still appear in the output of the command and the value of the *connected* attribute of the node will be "false".

The channel will remain open for a set blocktime, after which if the connection has not been re-established, the channel will close and the node will no longer appear in the command output.

32.3 RETURN VALUE

On success, an object with a "peers" key is returned containing a list of 0 or more objects.

Each object in the list contains the following data:

- *id* : The unique id of the peer
- *connected* : A boolean value showing the connection status
- *netaddr* : A list of network addresses the node is listening on
- *features* : Bit flags showing supported features (BOLT #9)
- *channels* : An list of channel id's open on the peer
- *log* : Only present if *level* is set. List logs related to the peer at the specified *level*

If *id* is supplied and no matching nodes are found, a "peers" object with an empty list is returned.

On error the returned object will contain `code` and `message` properties, with `code` being one of the following:

- -32602: If the given parameters are wrong.

32.4 AUTHOR

Michael Hawkins <michael.hawkins@protonmail.com>.

32.5 SEE ALSO

lightning-connect(7)

32.6 RESOURCES

Main web site: https://github.com/ElementsProject/lightning Lightning RFC site (BOLT #9): https://github.com/lightningnetwork/lightning-rfc/blob/master/09-features.md

CHAPTER 33

lightning-listsendpays – Low-level command for querying sendpay status

33.1 SYNOPSIS

listsendpays [*bolt11*] [*payment_hash*]

33.2 DESCRIPTION

The **listsendpays** RPC command gets the status of all *sendpay* commands (which is also used by the *pay* command), or with *bolt11* or *payment_hash* limits results to that specific payment. You cannot specify both.

Note that in future there may be more than one concurrent *sendpay* command per *pay*, so this command should be used with caution.

33.3 RETURN VALUE

On success, an array of objects is returned. Each object contains:

id unique internal value assigned at creation

payment_hash the hash of the *payment_preimage* which will prove payment.

destination the final destination of the payment.

amount_msat the amount the destination received, in "NNNmsat" format.

created_at the UNIX timestamp showing when this payment was initiated.

status one of *complete*, *failed* or *pending*.

payment_preimage (if *status* is *complete*) proves payment was received.

label optional *label*, if provided to *sendpay*.

bolt11 the *bolt11* argument given to *pay* (may be missing for pre-0.7 payments).

33.4 AUTHOR

Christian Decker <decker.christian@gmail.com> is mainly responsible.

33.5 SEE ALSO

lightning-listpays(7), lightning-sendpay(7), lightning-listinvoice(7).

33.6 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

CHAPTER 34

lightning-newaddr – Command for generating a new address to be used by c-lightning

34.1 SYNOPSIS

newaddr [*addresstype*]

34.2 DESCRIPTION

The **newaddr** RPC command generates a new address which can subsequently be used to fund channels managed by the c-lightning node.

The funding transaction needs to be confirmed before funds can be used.

addresstype specifies the type of address wanted; i.e. *p2sh-segwit* (e.g. 2MxaozoqWwiUcuD9KKgUSrLFDafLqimT9Ta on bitcoin testnet or 3MZxzq3jBSKNQ2e7dzneo9hy4FvNzmMmt3 on bitcoin mainnet) or *bech32 ‘ (e.g. tb1qu9j4lg5f9rgjyfhvfd905vw46eg39czmktxqgg on bitcoin testnet or bc1qwqdg6squsna38e46795at95yu9atm8azzmyvckulcc7kytlcckxswvvzej on bitcoin mainnet). The special value ‘all* generates both address types for the same underlying key.

If not specified the address generated is bech32.

34.3 RETURN VALUE

On success, a *bech32* address and/or a *p2sh-segwit* address will be returned.

34.4 ERRORS

If an unrecognized address type is requested an error message will be returned.

34.5 AUTHOR

Felix <fixone@gmail.com> is mainly responsible.

34.6 SEE ALSO

lightning-listfunds(7), lightning-fundchannel(7), lightning-withdraw(7)

34.7 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

CHAPTER 35

lightning-pay – Command for sending a payment to a BOLT11 invoice

35.1 SYNOPSIS

pay *bolt11* [*msatoshi*] [*label*] [*riskfactor*] [*maxfeepercent*] [*retry_for*] [*maxdelay*] [*exemptfee*]

35.2 DESCRIPTION

The **pay** RPC command attempts to find a route to the given destination, and send the funds it asks for. If the *bolt11* does not contain an amount, *msatoshi* is required, otherwise if it is specified it must be *null*. *msatoshi* is in millisatoshi precision; it can be a whole number, or a whole number with suffix *msat* or *sat*, or a three decimal point number with suffix *sat*, or an 1 to 11 decimal point number suffixed by *btc*.

The *label* field is used to attach a label to payments, and is returned in lightning-listpays(7) and lightning-listsendpays(7). The *riskfactor* is described in detail in lightning-getroute(7), and defaults to 10. The *maxfeepercent* limits the money paid in fees, and defaults to 0.5. The `maxfeepercent' is a percentage of the amount that is to be paid. The `exemptfee` option can be used for tiny payments which would be dominated by the fee leveraged by forwarding nodes. Setting `exemptfee` allows the `maxfeepercent` check to be skipped on fees that are smaller than `exemptfee` (default: 5000 millisatoshi).

The response will occur when the payment fails or succeeds. Once a payment has succeeded, calls to **pay** with the same *bolt11* will succeed immediately.

Until *retry_for* seconds passes (default: 60), the command will keep finding routes and retrying the payment. However, a payment may be delayed for up to `maxdelay` blocks by another node; clients should be prepared for this worst case.

When using *lightning-cli*, you may skip optional parameters by using *null*. Alternatively, use **-k** option to provide parameters by name.

35.3 RANDOMIZATION

To protect user privacy, the payment algorithm performs some randomization.

1: Route Randomization

Route randomization means the payment algorithm does not always use the lowest-fee or shortest route. This prevents some highly-connected node from learning all of the user payments by reducing their fees below the network average.

2: Shadow Route

Shadow route means the payment algorithm will virtually extend the route by adding delays and fees along it, making it appear to intermediate nodes that the route is longer than it actually is. This prevents intermediate nodes from reliably guessing their distance from the payee.

Route randomization will never exceed *maxfeepercent* of the payment. Route randomization and shadow routing will not take routes that would exceed *maxdelay*.

35.4 RETURN VALUE

On success, this returns the *payment_preimage* which hashes to the *payment_hash* to prove that the payment was successful. It will also return, a *getroute_tries* and a *sendpay_tries* statistics for the number of times it internally called **getroute** and **sendpay**.

You can monitor the progress and retries of a payment using the lightning-paystatus(7) command.

The following error codes may occur:

- -1: Catchall nonspecific error.
- 201: Already paid with this *hash* using different amount or destination.
- 203: Permanent failure at destination. The *data* field of the error will be routing failure object.
- 205: Unable to find a route.
- 206: Route too expensive. Either the fee or the needed total locktime for the route exceeds your *maxfeepercent* or *maxdelay* settings, respectively. The *data* field of the error will indicate the actual *fee* as well as the *feepercent* percentage that the fee has of the destination payment amount. It will also indicate the actual *delay* along the route.
- 207: Invoice expired. Payment took too long before expiration, or already expired at the time you initiated payment. The *data* field of the error indicates *now* (the current time) and *expiry* (the invoice expiration) as UNIX epoch time in seconds.
- 210: Payment timed out without a payment in progress.

Error codes 202 and 204 will only get reported at **sendpay**; in **pay** we will keep retrying if we would have gotten those errors.

A routing failure object has the fields below:

- *erring_index*: The index of the node along the route that reported the error. 0 for the local node, 1 for the first hop, and so on.
- *erring_node*: The hex string of the pubkey id of the node that reported the error.
- *erring_channel*: The short channel ID of the channel that has the error, or *0:0:0* if the destination node raised the error.
- *failcode*: The failure code, as per BOLT #4.

- *channel_update*. The hex string of the *channel_update* message received from the remote node. Only present if error is from the remote node and the *failcode* has the UPDATE bit set, as per BOLT #4.

The *data* field of errors will include statistics *getroute_tries* and *sendpay_tries*. It will also contain a *failures* field with detailed data about routing errors.

35.5 AUTHOR

Rusty Russell <rusty@rustcorp.com.au> is mainly responsible.

35.6 SEE ALSO

lightning-listpays(7), lightning-decodepay(7), lightning-listinvoice(7), lightning-delinvoice(7), lightning-getroute(7), lightning-invoice(7).

35.7 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

CHAPTER 36

lightning-plugin – Manage plugins with RPC

36.1 SYNOPSIS

plugin command [parameter] [second_parameter]

36.2 DESCRIPTION

The **plugin** RPC command allows to manage plugins without having to restart lightningd. It takes 1 to 3 parameters: a command (start/stop/startdir/rescan/list) which describes the action to take and optionally one or two parameters which describes the plugin on which the action has to be taken.

The *start* command takes a path as the first parameter and will load the plugin available from this path. It will wait for the plugin to complete the handshake with `lightningd` for 20 seconds at the most.

The *stop* command takes a plugin name as parameter. It will kill and unload the specified plugin.

The *startdir* command takes a directory path as first parameter and will load all plugins this directory contains. It will wait for each plugin to complete the handshake with `lightningd` for 20 seconds at the most.

The *rescan* command starts all not-already-loaded plugins from the default plugins directory (by default *~/.lightning/plugins*).

The *list* command will return all the active plugins.

36.3 RETURN VALUE

On success, all subcommands but *stop* return an array *plugins* of objects, one by plugin. Each object contains the name of the plugin (*name* field) and its status (*active* boolean field). Since plugins are configured asynchronously, a freshly started plugin may not appear immediately.

On error, the reason why the action could not be taken upon the plugin is returned.

36.4 AUTHOR

Antoine Poinsot <darosior@protonmail.com> is mainly responsible.

36.5 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

CHAPTER 37

lightning-sendonion – Send a payment with a custom onion packet

37.1 SYNOPSIS

sendonion *onion first_hop payment_hash* [*label*] [*shared_secrets*]

37.2 DESCRIPTION

The **sendonion** RPC command can be used to initiate a payment attempt with a custom onion packet. The onion packet is used to deliver instructions for hops along the route on how to behave. Normally these instructions are indications on where to forward a payment and what parameters to use, or contain details of the payment for the final hop. However, it is possible to add arbitrary information for hops in the custom onion, allowing for custom extensions that are not directly supported by c-lightning.

The onion is specific to the route that is being used and the *payment_hash* used to construct, and therefore cannot be reused for other payments or to attempt a separate route. The custom onion can generally be created using the `devtools/onion` CLI tool, or the **createonion** RPC command.

The *onion* parameter is a hex-encoded 1366 bytes long blob that was returned by either of the tools that can generate onions. It contains the payloads destined for each hop and some metadata. Please refer to BOLT 04 for further details.

The *first_hop* parameter instructs c-lightning which peer to send the onion to. It is a JSON dictionary that corresponds to the first element of the route array returned by *getroute*. The following is a minimal example telling c-lightning to use channel `103x1x1` to add an HTLC for 1002 millisatoshis and a delay of 21 blocks on top of the current blockheight:

```
{
  "channel": "103x1x1",
  "direction": 1,
  "amount_msat": "1002msat",
  "delay": 21,
}
```

The *payment_hash* parameter specifies the 32 byte hex-encoded hash to use as a challenge to the HTLC that we are sending. It is specific to the onion and has to match the one the onion was created with.

The *label* parameter can be used to provide a human readable reference to retrieve the payment at a later time.

The *shared_secrets* parameter is a JSON list of 32 byte hex-encoded secrets that were used when creating the onion. c-lightning can send a payment with a custom onion without the knowledge of these secrets, however it will not be able to parse an eventual error message since that is encrypted with the shared secrets used in the onion. If *shared_secrets* is provided c-lightning will decrypt the error, act accordingly, e.g., add a `channel_update` included in the error to its network view, and set the details in *listsendpays* correctly. If it is not provided c-lightning will store the encrypted onion, and expose it in *listsendpays* allowing the caller to decrypt it externally. The following is an example of a 3 hop onion:

```
[
        "298606954e9de3e9d938d18a74fed794c440e8eda82e52dc08600953c8acf9c4",
        "2dc094de72adb03b90894192edf9f67919cb2691b37b1f7d4a2f4f31c108b087",
        "a7b82b240dbd77a4ac8ea07709b1395d8c510c73c17b4b392bb1f0605d989c85"
]
```

If *shared_secrets* is not provided the c-lightning node does not know how long the route is, which channels or nodes are involved, and what an eventual error could have been. It can therefore be used for oblivious payments.

37.3 RETURN VALUE

On success, an object similar to the output of **sendpay** will be returned. This object will have a *status* field that is typically the string *"pending"*, but may be *"complete"* if the payment was already performed successfully.

If *shared_secrets* was provided and an error was returned by one of the intermediate nodes the error details are decrypted and presented here. Otherwise the error code is 202 for an unparseable onion.

37.4 AUTHOR

Christian Decker <decker.christian@gmail.com> is mainly responsible.

37.5 SEE ALSO

lightning-createonion(7), lightning-sendpay(7), lightning-listsendpays(7)

37.6 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

CHAPTER 38

lightning-sendpay – Low-level command for sending a payment via a route

38.1 SYNOPSIS

sendpay *route payment_hash* [*label*] [*msatoshi*] [*bolt11*] [*partid*]

38.2 DESCRIPTION

The **sendpay** RPC command attempts to send funds associated with the given *payment_hash*, along a route to the final destination in the route.

Generally, a client would call lightning-getroute(7) to resolve a route, then use **sendpay** to send it. If it fails, it would call lightning-getroute(7) again to retry.

The response will occur when the payment is on its way to the destination. The **sendpay** RPC command does not wait for definite success or definite failure of the payment. Instead, use the **waitsendpay** RPC command to poll or wait for definite success or definite failure.

The *label* and *bolt11* parameters, if provided, will be returned in *waitsendpay* and *listsendpays* results.

The *msatoshi* amount, if provided, is the amount that will be recorded as the target payment value. If not specified, it will be the final amount to the destination. By default it is in millisatoshi precision; it can be a whole number, or a whole number ending in *msat* or *sat*, or a number with three decimal places ending in *sat*, or a number with 1 to 11 decimal places ending in *btc*.

The *partid* value, if provided and non-zero, allows for multiple parallel partial payments with the same *payment_hash*. The *msatoshi* amount (which must be provided) for each **sendpay** with matching *payment_hash* must be equal, and **sendpay** will fail if there are already *msatoshi* worth of payments pending.

Once a payment has succeeded, calls to **sendpay** with the same *payment_hash* but a different *msatoshi* or destination will fail; this prevents accidental multiple payments. Calls to **sendpay** with the same *payment_hash*, *msatoshi*, and destination as a previous successful payment (even if a different route or *partid*) will return immediately with success.

38.3 RETURN VALUE

On success, an object similar to the output of **listsendpays** will be returned. This object will have a *status* field that is typically the string *"pending"*, but may be *"complete"* if the payment was already performed successfully.

On error, if the error occurred from a node other than the final destination, the route table will be updated so that lightning-getroute(7) should return an alternate route (if any). An error from the final destination implies the payment should not be retried.

The following error codes may occur:

- -1: Catchall nonspecific error.
- 201: Already paid with this *hash* using different amount or destination.
- 202: Unparseable onion reply. The *data* field of the error will have an *onionreply* field, a hex string representation of the raw onion reply.
- 203: Permanent failure at destination. The *data* field of the error will be routing failure object.
- 204: Failure along route; retry a different route. The *data* field of the error will be routing failure object.

A routing failure object has the fields below:

- *erring_index*. The index of the node along the route that reported the error. 0 for the local node, 1 for the first hop, and so on.
- *erring_node*. The hex string of the pubkey id of the node that reported the error.
- *erring_channel*. The short channel ID of the channel that has the error, or *0:0:0* if the destination node raised the error.
- *failcode*. The failure code, as per BOLT #4.
- *channel_update*. The hex string of the *channel_update* message received from the remote node. Only present if error is from the remote node and the *failcode* has the UPDATE bit set, as per BOLT #4.

38.4 AUTHOR

Rusty Russell <rusty@rustcorp.com.au> is mainly responsible.

38.5 SEE ALSO

lightning-listinvoice(7), lightning-delinvoice(7), lightning-getroute(7), lightning-invoice(7), lightning-pay(7), lightning-waitsendpay(7).

38.6 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

CHAPTER 39

lightning-setchannelfee – Command for setting specific routing fees on a lightning channel

39.1 SYNOPSIS

setchannelfee *id* [*base*] [*ppm*]

39.2 DESCRIPTION

The **setchannelfee** RPC command sets channel specific routing fees as defined in BOLT #7. The channel has to be in normal or awaiting state. This can be checked by **listpeers** reporting a *state* of CHANNELD_NORMAL or CHANNELD_AWAITING_LOCKIN for the channel.

id is required and should contain a scid (short channel ID), channel id or peerid (pubkey) of the channel to be modified. If *id* is set to "all", the fees for all channels are updated that are in state CHANNELD_NORMAL or CHANNELD_AWAITING_LOCKIN.

base is an optional value in millisatoshi that is added as base fee to any routed payment. If the parameter is left out, the global config value fee-base will be used again. It can be a whole number, or a whole number ending in *msat* or *sat*, or a number with three decimal places ending in *sat*, or a number with 1 to 11 decimal places ending in *btc*.

ppm is an optional value that is added proportionally per-millionths to any routed payment volume in satoshi. For example, if ppm is 1,000 and 1,000,000 satoshi is being routed trhough the channel, an proportional fee of 1,000 satoshi is added, resulting in a 0.1% fee. If the parameter is left out, the global config value will be used again.

39.3 RETURN VALUE

On success, an object with the new values *base* and *ppm* along with an array *channels* which contains objects with fields *peer_id*, *channel_id* and *short_channel_id*.

39.4 ERRORS

The following error codes may occur:

- -1: Channel is in incorrect state, i.e. Catchall nonspecific error.
- -32602: JSONRPC2_INVALID_PARAMS, i.e. Given id is not a channel ID or short channel ID.

39.5 AUTHOR

Michael Schmoock <michael@schmoock.net> is the author of this feature. Rusty Russell <rusty@rustcorp.com.au> is mainly responsible for the c-lightning project.

39.6 SEE ALSO

lightningd-config(5), lightning-fundchannel(7), lightning-listchannels(7), lightning-listpeers(7)

39.7 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

CHAPTER 40

lightning-signmessage – Command to create a signature from this node

40.1 SYNOPSIS

signmessage *message*

40.2 DESCRIPTION

The **signmessage** RPC command creates a digital signature of *message* using this node's secret key. A receiver who knows your node's *id* and the *message* can be sure that the resulting signature could only be created by something with access to this node's secret key.

message must be less that 65536 characters.

40.3 RETURN VALUE

An object with attributes *signature*, *recid* and *zbase* is returned. *zbase* is the result of *signature* and *recid* encoded in a style compatible with **lnd**'s SignMessageRequest.

40.4 AUTHOR

Rusty Russell <rusty@rustcorp.com.au> is mainly responsible.

40.5 SEE ALSO

lightning-checkmessage(7)

40.6 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

CHAPTER 41

lightning-txdiscard – Abandon a transaction from txprepare, release inputs

41.1 SYNOPSIS

txdiscard *txid*

41.2 DESCRIPTION

The **txdiscard** RPC command releases inputs which were reserved for use of the *txid* from lightning-txprepare(7).

41.3 RETURN VALUE

On success, an object with attributes *unsigned_tx* and *txid* will be returned, exactly as from lightning-txprepare(7).

If there is no matching *txid*, an error is reported. Note that this may happen due to incorrect usage (such as **txdiscard** or **txsend** already being called for *txid*) or due to lightningd restarting, which implicitly calls **txdiscard** on all outputs.

The following error codes may occur:

- -1: An unknown *txid*.

41.4 AUTHOR

Rusty Russell <rusty@rustcorp.com.au> is mainly responsible.

41.5 SEE ALSO

lightning-txprepare(7), lightning-txsend(7)

41.6 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

CHAPTER 42

lightning-txprepare – Command to prepare to withdraw funds from the internal wallet

42.1 SYNOPSIS

txprepare *outputs* [*feerate*] [*minconf*] [*utxos*]

42.2 DESCRIPTION

The **txprepare** RPC command creates an unsigned transaction which spends funds from c-lightning's internal wallet to the outputs specified in *outputs*.

The *outputs* is the array of output that include *destination* and *amount*({*destination*: *amount*}). Its format is like: [{address1: amount1}, {address2: amount2}] or [{address: *all*}]. It supports the any number of outputs.

The *destination* of output is the address which can be of any Bitcoin accepted type, including bech32.

The *amount* of output is the amount to be sent from the internal wallet (expressed, as name suggests, in amount). The string *all* can be used to specify all available funds. Otherwise, it is in amount precision; it can be a whole number, a whole number ending in *sat*, a whole number ending in *000msat*, or a number with 1 to 8 decimal places ending in *btc*.

feerate is an optional feerate to use. It can be one of the strings *urgent* (aim for next block), *normal* (next 4 blocks or so) or *slow* (next 100 blocks or so) to use lightningd's internal estimates: *normal* is the default.

Otherwise, *feerate* is a number, with an optional suffix: *perkw* means the number is interpreted as satoshi-per-kilosipa (weight), and *perkb* means it is interpreted bitcoind-style as satoshi-per-kilobyte. Omitting the suffix is equivalent to *perkb*.

minconf specifies the minimum number of confirmations that used outputs should have. Default is 1.

utxos specifies the utxos to be used to fund the transaction, as an array of "txid:vout". These must be drawn from the node's available UTXO set.

txprepare is similar to the first part of a **withdraw** command, but supports multiple outputs and uses *outputs* as parameter. The second part is provided by **txsend**.

42.3 RETURN VALUE

On success, an object with attributes *unsigned_tx* and *txid* will be returned. You need to hand *txid* to **txsend** or **txdiscard**, as the inputs of this transaction are reserved until then, or until the daemon restarts.

unsigned_tx represents the raw bitcoin transaction (not yet signed) and *txid* represent the bitcoin transaction id.

On failure, an error is reported and the transaction is not created.

The following error codes may occur:

- -1: Catchall nonspecific error.
- 301: There are not enough funds in the internal wallet (including fees) to create the transaction.
- 302: The dust limit is not met.

42.4 AUTHOR

Rusty Russell <rusty@rustcorp.com.au> is mainly responsible.

42.5 SEE ALSO

lightning-withdraw(7), lightning-txsend(7), lightning-txdiscard(7)

42.6 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

CHAPTER 43

lightning-txsend – Command to sign and send transaction from txprepare

43.1 SYNOPSIS

txsend *txid*

43.2 DESCRIPTION

The **txsend** RPC command signs and broadcasts a transaction created by **txprepare**.

43.3 RETURN VALUE

On success, an object with attributes *tx* and *txid* will be returned.

tx represents the fully signed raw bitcoin transaction, and *txid* is the same as the *txid* argument.

On failure, an error is reported (from bitcoind), and the inputs from the transaction are unreserved.

The following error codes may occur:

- -1: Catchall nonspecific error.

43.4 AUTHOR

Rusty Russell <rusty@rustcorp.com.au> is mainly responsible.

43.5 SEE ALSO

lightning-txprepare(7), lightning-txdiscard(7)

43.6 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

CHAPTER 44

lightning-waitanyinvoice – Command for waiting for payments

44.1 SYNOPSIS

waitanyinvoice [*lastpay_index*]

44.2 DESCRIPTION

The **waitanyinvoice** RPC command waits until an invoice is paid, then returns a single entry as per **listinvoice**. It will not return for any invoices paid prior to or including the *lastpay_index*.

This is usually called iteratively: once with no arguments, then repeatedly with the returned *pay_index* entry. This ensures that no paid invoice is missed.

The *pay_index* is a monotonically-increasing number assigned to an invoice when it gets paid. The first valid *pay_index* is 1; specifying *lastpay_index* of 0 equivalent to not specifying a *lastpay_index*. Negative *lastpay_index* is invalid.

44.3 RETURN VALUE

On success, an invoice description will be returned as per lightning-listinvoice(7): *complete* will always be *true*.

44.4 AUTHOR

Rusty Russell <rusty@rustcorp.com.au> is mainly responsible.

44.5 SEE ALSO

lightning-waitinvoice(7), lightning-listinvoice(7), lightning-delinvoice(7), lightning-invoice(7).

44.6 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

CHAPTER 45

lightning-waitblockheight – Command for waiting for blocks on the blockchain

45.1 SYNOPSIS

waitblockheight *blockheight* [*timeout*]

45.2 DESCRIPTION

The **waitblockheight** RPC command waits until the blockchain has reached the specified *blockheight*. It will only wait up to *timeout* seconds (default 60).

If the *blockheight* is a present or past block height, then this command returns immediately.

45.3 RETURN VALUE

Once the specified block height has been achieved by the blockchain, an object with the single field *blockheight* is returned, which is the block height at the time the command returns.

If *timeout* seconds is reached without the specified blockheight being reached, this command will fail.

45.4 AUTHOR

ZmnSCPxj <ZmnSCPxj@protonmail.com> is mainly responsible.

45.5 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

CHAPTER 46

lightning-waitinvoice – Command for waiting for specific payment

46.1 SYNOPSIS

waitinvoice *label*

46.2 DESCRIPTION

The **waitinvoice** RPC command waits until a specific invoice is paid, then returns that single entry as per **listinvoice**.

46.3 RETURN VALUE

On success, an invoice description will be returned as per lightning-listinvoice(7). The *status* field will be *paid*.

If the invoice is deleted while unpaid, or the invoice does not exist, this command will return with an error with code -1.

If the invoice expires before being paid, or is already expired, this command will return with an error with code -2, with the data being the invoice data as per **listinvoice**.

46.4 AUTHOR

Christian Decker <decker.christian@gmail.com> is mainly responsible.

46.5 SEE ALSO

lightning-waitanyinvoice(7), lightning-listinvoice(7), lightning-delinvoice(7), lightning-invoice(7)

46.6 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

CHAPTER 47

lightning-waitsendpay – Command for sending a payment via a route

47.1 SYNOPSIS

waitsendpay *payment_hash* [*timeout*] [*partid*]

47.2 DESCRIPTION

The **waitsendpay** RPC command polls or waits for the status of an outgoing payment that was initiated by a previous **sendpay** invocation.

The *partid* argument must match that of the **sendpay** command.

Optionally the client may provide a *timeout*, an integer in seconds, for this RPC command to return. If the *timeout* is provided and the given amount of time passes without the payment definitely succeeding or definitely failing, this command returns with a 200 error code (payment still in progress). If *timeout* is not provided this call will wait indefinitely.

Indicating a *timeout* of 0 effectively makes this call a pollable query of the status of the payment.

If the payment completed with success, this command returns with success. Otherwise, if the payment completed with failure, this command returns an error.

47.3 RETURN VALUE

On success, an object similar to the output of **listsendpays** will be returned. This object will have a *status* field that is the string *"complete"*.

On error, and even if the error occurred from a node other than the final destination, the route table will no longer be updated. Use the *exclude* parameter of the `getroute` command to ignore the failing route.

The following error codes may occur:

- -1: Catchall nonspecific error.
- 200: Timed out before the payment could complete.
- 202: Unparseable onion reply. The *data* field of the error will have an *onionreply* field, a hex string representation of the raw onion reply.
- 203: Permanent failure at destination. The *data* field of the error will be routing failure object.
- 204: Failure along route; retry a different route. The *data* field of the error will be routing failure object.
- 208: A payment for *payment_hash* was never made and there is nothing to wait for.
- 209: The payment already failed, but the reason for failure was not stored. This should only occur when querying failed payments on very old databases.

A routing failure object has the fields below:

- *erring_index*: The index of the node along the route that reported the error. 0 for the local node, 1 for the first hop, and so on.
- *erring_node*: The hex string of the pubkey id of the node that reported the error.
- *erring_channel*: The short channel ID of the channel that has the error (or the final channel if the destination raised the error).
- *erring_direction*: The direction of traversing the *erring_channel*:
- *failcode*: The failure code, as per BOLT #4.
- *failcodename*: The human-readable name corresponding to *failcode*, if known.

47.4 AUTHOR

ZmnSCPxj <ZmnSCPxj@protonmail.com> is mainly responsible.

47.5 SEE ALSO

lightning-sendpay(7), lightning-pay(7).

47.6 RESOURCES

Main web site: https://github.com/ElementsProject/lightning

CHAPTER 48

lightning-withdraw – Command for withdrawing funds from the internal wallet

48.1 SYNOPSIS

withdraw *destination satoshi* [*feerate*] [*minconf*] [*utxos*]

48.2 DESCRIPTION

The **withdraw** RPC command sends funds from c-lightning's internal wallet to the address specified in *destination*.

The address can be of any Bitcoin accepted type, including bech32.

satoshi is the amount to be withdrawn from the internal wallet (expressed, as name suggests, in satoshi). The string *all* can be used to specify withdrawal of all available funds. Otherwise, it is in satoshi precision; it can be a whole number, a whole number ending in *sat*, a whole number ending in *000msat*, or a number with 1 to 8 decimal places ending in *btc*.

feerate is an optional feerate to use. It can be one of the strings *urgent* (aim for next block), *normal* (next 4 blocks or so) or *slow* (next 100 blocks or so) to use lightningd's internal estimates: *normal* is the default.

Otherwise, *feerate* is a number, with an optional suffix: *perkw* means the number is interpreted as satoshi-per-kilosipa (weight), and *perkb* means it is interpreted bitcoind-style as satoshi-per-kilobyte. Omitting the suffix is equivalent to *perkb*.

minconf specifies the minimum number of confirmations that used outputs should have. Default is 1.

utxos specifies the utxos to be used to be withdrawn from, as an array of "txid:vout". These must be drawn from the node's available UTXO set.

48.3 RETURN VALUE

On success, an object with attributes *tx* and *txid* will be returned.

tx represents the raw bitcoin, fully signed, transaction and *txid* represent the bitcoin transaction id.

On failure, an error is reported and the withdrawal transaction is not created.

The following error codes may occur:

- -1: Catchall nonspecific error.
- 301: There are not enough funds in the internal wallet (including fees) to create the transaction.
- 302: The dust limit is not met.

48.4 AUTHOR

Felix <fixone@gmail.com> is mainly responsible.

48.5 SEE ALSO

lightning-listfunds(7), lightning-fundchannel(7), lightning-newaddr(7), lightning-txprepare(7).

48.6 RESOURCES

Main web site: https://github.com/ElementsProject/lightning