core-lightning

Release v22.11rc1-modded

November 11, 2022

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` with `ECONOMICAL` mode support, 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 \
  python3 python3-pip net-tools zlib1g-dev libsodium-dev gettext
pip3 install --upgrade pip
pip3 install --user poetry
```

If you don't have Bitcoin installed locally you'll need to install that as well. It's now available via snapd.

```
sudo apt-get install snapd
sudo snap install bitcoin-core
# Snap does some weird things with binary names; you'll
# want to add a link to them so everything works as expected
sudo ln -s /snap/bitcoin-core/current/bin/bitcoin{d,-cli} /usr/local/bin/
```

Clone lightning:

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

Checkout a release tag:

```
git checkout v0.11.2
```

For development or running tests, get additional dependencies:

```
sudo apt-get install -y valgrind libpq-dev shellcheck cppcheck \
  libsecp256k1-dev jq lowdown
```

If you can't install `lowdown`, a version will be built in-tree.

If you want to build the Rust plugins (currently, cln-grpc):

```
sudo apt-get install -y cargo rustfmt
```

There are two ways to build core lightning, and this depends on how you want use it.

To build cln to just install a tagged or master version you can use the following commands:

```
pip3 install --upgrade pip
pip3 install mako
./configure
make
sudo make install
```

N.B: if you want disable Rust because you do not want use it or simple you do not want the grpc-plugin, you can use `./configure --disable-rust`.

To build core lightning for development purpose you can use the following commands:

```
pip3 install poetry
poetry shell
```

This will put you in a new shell to enter the following commands:

```
poetry install
./configure --enable-developer
make
make check VALGRIND=0
```

optionally, add `-j$(nproc)` after `make` to speed up compilation. (e.g. `make -j$(nproc)`)

Running lightning:

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

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 \
                gettext \
                git \
                gmp-devel \
                libsq3-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
```

Checkout a release tag:

```
$ git checkout v0.11.2
```

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

Core 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
```

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 OpenBSD

OS version: OpenBSD 6.7

Install dependencies:

```
pkg_add git python gmake py3-pip libtool gmp
pkg_add automake # (select highest version, automake1.16.2 at time of writing)
pkg_add autoconf # (select highest version, autoconf-2.69p2 at time of writing)
```

Install `mako` otherwise we run into build errors:

```
pip3.7 install --user poetry
poetry install
```

Add `/home/<username>/.local/bin` to your path:

`export PATH=$PATH:/home/<username>/.local/bin`

Needed for `configure`:

```
export AUTOCONF_VERSION=2.69
export AUTOMAKE_VERSION=1.16
./configure
```

Finally, build `c-lightning`:

`gmake`

1.6 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
```

1.7 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"
```

Some library paths are different when using `homebrew` with M1 macs, therefore the following two variables need to be set for M1 machines

```
$ export CPATH=/opt/homebrew/include
$ export LIBRARY_PATH=/opt/homebrew/lib
```

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
$ pip install poetry
```

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
```

Checkout a release tag:

```
$ git checkout v0.11.2
```

Build lightning:

```
$ poetry install
$ ./configure
$ poetry run 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
```

To install the built binaries into your system, you'll need to run `make install`:

```
make install
```

On an M1 mac you may need to use this command instead:

```
sudo PATH="/usr/local/opt:$PATH"  LIBRARY_PATH=/opt/homebrew/lib CPATH=/opt/homebrew/
↪include make install
```

1.8 To Build on Arch Linux

Install dependencies:

```
pacman --sync autoconf automake gcc git make python-pip
pip install --user poetry
```

Clone Core Lightning:

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

Build Core Lightning:

```
python -m poetry install
./configure
python -m poetry run make
```

Launch Core Lightning:

```
./lightningd/lightningd
```

1.9 To cross-compile for Android

Make a standalone toolchain as per https://developer.android.com/ndk/guides/standalone_toolchain.html. For Core 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"
```

1.10 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-1.2.12.tar.gz
tar xvf zlib-1.2.12.tar.gz
cd zlib-1.2.12
./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
make
make install
```

Then, build Core Lightning with the following commands:

```
./configure
make
```

1.11 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 Core Lightning for Armbian can be found here.

1.12 To compile for Alpine

Get dependencies:

```
apk update
apk add --virtual .build-deps ca-certificates alpine-sdk autoconf automake git␣
↪libtool \
  gmp-dev sqlite-dev python3 py3-mako net-tools zlib-dev libsodium gettext
```

Clone lightning:

```
git clone https://github.com/ElementsProject/lightning.git
cd lightning
git submodule update --init --recursive
```

Build and install:

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

Clean up:

```
cd .. && rm -rf lightning
apk del .build-deps
```

Install runtime dependencies:

```
apk add gmp libgcc libsodium sqlite-libs zlib
```

1.13 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 Core Lightning

To use any Tor features with Core Lightning you must have Tor installed and running.

Note that we only support Tor v3: you can check your installed Tor version with `tor --version` or `sudo tor --version`

If Tor is not installed you can install it on Debian based Linux systems (Ubuntu, Debian, etc) with the following command:

```
sudo apt install tor
```

then `/etc/init.d/tor start` or `sudo systemctl enable --now 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 Core 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 Core 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 Quick Start On Linux

It is easy to create a single persistent Tor address and not announce a public IP. This is ideal for most setups where you have an ISP-provided router connecting your Internet to your local network and computer, as it does not require a stable public IP from your ISP (which might not give one to you for free), nor port forwarding (which can be hard to set up for random cheap router models). Tor provides NAT-traversal for free, so even if you or your ISP has a complex network between you and the Internet, as long as you can use Tor you can be connected to.

Note: Core Lightning also support IPv4/6 address discovery behind NAT routers. For this to work you need to forward the default TCP port 9735 to your node. In this case you don't need TOR to punch through your firewall. IP discovery is only active if no other addresses are announced. This usually has the benefit of quicker and more stable connections but does not offer additional privacy.

On most Linux distributions, making a standard installation of `tor` will automatically set it up to have a SOCKS5 proxy at port 9050. As well, you have to set up the Tor Control Port. On most Linux distributions there will be commented-out settings below in the `/etc/tor/torrc`:

```
ControlPort 9051
CookieAuthentication 1
CookieAuthFile /var/lib/tor/control_auth_cookie
CookieAuthFileGroupReadable 1
```

Uncomment those in, then restart `tor` (usually `systemctl restart tor` or `sudo systemctl restart tor` on most SystemD-based systems, including recent Debian and Ubuntu, or just restart the entire computer if you cannot figure it out).

On some systems (such as Arch Linux), you may also need to add the following setting:

```
DataDirectoryGroupReadable 1
```

You also need to make your user a member of the Tor group. "Your user" here is whatever user will run `lightningd`. On Debian-derived systems, the Tor group will most likely be `debian-tor`. You can try listing all groups with the below command, and check for a `debian-tor` or `tor` groupname.

```
getent group | cut -d: -f1 | sort
```

Alternately, you could check the group of the cookie file directly. Usually, on most Linux systems, that would be `/run/tor/control.authcookie`:

```
stat -c '%G' /run/tor/control.authcookie
```

Once you have determined the `${TORGROUP}` and selected the `${LIGHTNINGUSER}` that will run `lightningd`, run this as root:

```
usermod -a -G ${TORGROUP} ${LIGHTNINGUSER}
```

Then restart the computer (logging out and logging in again should also work). Confirm that `${LIGHTNINGUSER}` is in `${TORGROUP}` by running the `groups` command as `${LIGHTNINGUSER}` and checking `${TORGROUP}` is listed.

If the `/run/tor/control.authcookie` exists in your system, then log in as the user that will run `lightningd` and check this command:

```
cat /run/tor/control.authcookie > /dev/null
```

If the above prints nothing and returns, then Core Lightning "should" work with your Tor. If it prints an error, some configuration problem will likely prevent Core Lightning from working with your Tor.

Then make sure these are in your `${LIGHTNING_DIR}/config` or other Core Lightning configuration (or prepend `--` to each of them and add them to your `lightningd` invocation command line):

```
proxy=127.0.0.1:9050
bind-addr=127.0.0.1:9735
addr=statictor:127.0.0.1:9051
always-use-proxy=true
```

1. `proxy` informs Core Lightning that you have a SOCKS5 proxy at port 9050. Core Lightning will assume that this is a Tor proxy, port 9050 is the default in most Linux distributions; you can double-check `/etc/tor/torrc` for a `SocksPort` entry to confirm the port number.
2. `bind-addr` informs Core Lightning to bind itself to port 9735. This is needed for the subsequent `statictor` to work. 9735 is the normal Lightning Network port, so this setting may already be present. If you add a second `bind-addr=...` you may get errors, so choose this new one or keep the old one, but don't keep both. This has to appear before any `statictor:` setting.
3. `addr=statictor:` informs Core Lightning that you want to create a persistent hidden service that is based on your node private key. This informs Core Lightning as well that the Tor Control Port is 9051. You can also use `bind-addr=statictor:` instead to not announce the persistent hidden service, but if anyone wants to make a channel with you, you either have to connect to them, or you have to reveal your address to them explicitly (i.e. autopilots and the like will likely never connect to you).
4. `always-use-proxy` informs Core Lightning to always use Tor even when connecting to nodes with public IPs. You can set this to `false` or remove it, if you are not privacy-conscious **and** find Tor is too slow for you.

2.2 Tor Browser and Orbot

It is possible to not install Tor on your computer, and rely on just Tor Browser. Tor Browser will run a built-in Tor instance, but with the proxy at port 9150 and the control port at 9151 (the normal Tor has, by default, the proxy at port 9050 and the control port at 9051). The mobile Orbot uses the same defaults as Tor Browser (9150 and 9151).

You can then use these settings for Core Lightning:

```
proxy=127.0.0.1:9150
bind-addr=127.0.0.1:9735
addr=statictor:127.0.0.1:9151
always-use-proxy=true
```

You will have to run Core Lightning after launching Tor Browser or Orbot, and keep Tor Browser or Orbot open as long as Core Lightning is running, but this is a setup which allows others to connect and fund channels to you, anywhere (no port forwarding! works wherever Tor works!), and you do not have to do anything more complicated than download and install Tor Browser. This may be useful for operating system distributions that do not have Tor in their repositories, assuming we can ever get Core Lightning running on those.

2.3 Detailed Discussion

2.3.1 Three Ways to Create .onion Addresses for Core Lightning

1. You can configure Tor to create an onion address for you, and tell Core Lightning to use that address
2. You can have Core Lightning tell Tor to create a new onion address every time
3. You can configure Core Lightning to tell Tor to create the same onion address every time it starts up

2.3.2 Tor-Created .onion Address

Having Tor create an onion address lets you run other services (e.g. a web server) at that same address, and you just tell that address to Core Lightning and it doesn't have to talk to the Tor server at all.

Put the following in your `/etc/tor/torrc` file:

```
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 restart tor` depending on the configuration of your system.

You will find the newly created address (myaddress.onion) with:

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

Now you need to tell Core Lightning to advertize that onion hostname and port, by placing `announce-addr=myaddress.onion` in your lightning config.

2.3.3 Letting Core Lightning Control Tor

To have Core Lightning control your Tor addresses, you have to tell Tor to accept control commands from Core Lightning, either by using a cookie, or a password.

Service authenticated by cookie

This tells Tor to create a cookie file each time: lightningd will have to be in the same group as tor (e.g. debian-tor): you can look at `/run/tor/control.authcookie` to check the group name.

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

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

Save the file and restart the Tor service.

Service authenticated by password

This tells Tor to allow password access: you also need to tell lightningd what the password is.

Create a hash of your password with

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

This returns a line like

`16:533E3963988E038560A8C4EE6BBEE8DB106B38F9C8A7F81FE38D2A3B1F`

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

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

Save the file and restart the Tor service.

Put `tor-service-password=yourpassword` (not the hash) in your lightning configuration file.

Core Lightning Creating Persistent Hidden Addresses

This is usually better than transient addresses, as nodes won't have to wait for gossip propagation to find out your new address each time you restart.

Once you've configured access to Tor as described above, you need to add *two* lines in your lightningd config file:

1. A local address which lightningd can tell Tor to connect to when connections come in, e.g. `bind-addr=127.0.0.1:9735`.
2. After that, a `addr=statictor:127.0.0.1:9051` to tell Core Lightning to set up and announce a Tor onion address (and tell Tor to send connections to our real address, above).

You can use `bind-addr` if you want to set up the onion address and not announce it to the world for some reason.

You may add more `addr` lines if you want to advertize other addresses.

There is an older method, called "autotor" instead of "statictor" which creates a different Tor address on each restart, which is usually not very helpful; you need to use `lightning-cli getinfo` to see what address it is currently using, and other peers need to wait for fresh gossip messages if you announce it, before they can connect.

2.4 What do we support

| Case # | IP Number | Hidden service |Incoming / Outgoing Tor | | ——- | ————- | —————————- |—————————- | 1 | Public | NO | Outgoing | | 2 | Public | FIXED BY TOR | Incoming [1] | | 3 | Public | FIXED BY CORE LIGHTNING | Incoming [1] | | 4 | Not Announced | FIXED BY TOR | Incoming [1] | | 5 | Not Announced | FIXED BY CORE LIGHTNING | Incoming [1] |

NOTE:

1. 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.

2.4.1 Case #1: Public IP address and no Tor address, but can connect to Tor addresses

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 (though if you only advertize Tor addresses, we also assume you want to always use the proxy).

You can announce your public IP address through the usual method: if your node is in an internal network:

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

or if it has a public IP address:

```
addr=externalIpAddress
```

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.

2.4.2 Case #2: Public IP address, and a fixed Tor address in torrc

Other nodes can connect to you entirely over Tor, and the Tor address doesn't change every time you restart.

You simply tell Core Lightning to advertize both addresses (you can use `sudo cat /var/lib/tor/lightningd-service_v3/hostname` to get your Tor-assigned onion address).

If you have an internal IP address:

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

Or an external address:

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

2.4.3 Case #3: Public IP address, and a fixed Tor address set by Core Lightning

Other nodes can connect to you entirely over Tor, and the Tor address doesn't change every time you restart.

See "Letting Core Lightning Control Tor" for how to get Core Lightning talking to Tor.

If you have an internal IP address:

```
bind-addr=yourInternalIPAddress:port
announce-addr=yourexternalIPAddress:port
addr=statictor:127.0.0.1:9051
```

Or an external address:

```
addr=yourIPAddress:port
addr=statictor:127.0.0.1:9051
```

2.4.4 Case #4: Unannounced IP address, and a fixed Tor address in torrc

Other nodes can only connect to you over Tor.

You simply tell Core Lightning to advertize the Tor address (you can use `sudo cat /var/lib/tor/lightningd-service_v3/hostname` to get your Tor-assigned onion address).

```
announce-addr=your.onionAddress:port
proxy=127.0.0.1:9050
always-use-proxy=true
```

2.4.5 Case #4: Unannounced IP address, and a fixed Tor address set by Core Lightning

Other nodes can only connect to you over Tor.

See "Letting Core Lightning Control Tor" for how to get Core Lightning talking to Tor.

```
addr=statictor:127.0.0.1:9051
proxy=127.0.0.1:9050
always-use-proxy=true
```

2.5 References

The lightningd-config manual page covers the various address cases in detail.

The Tor project

CHAPTER 3

FAQ

3.1 Table of contents

3.2 General questions

3.2.1 I don't know where to start, help me !

There is a C-lightning plugin specifically for this purpose, it's called `helpme`.

Assuming you have followed the *installation steps*, have `lightningd` up and running, and `lightning-cli` in your `$PATH` you can start the plugin like so:

```
# Clone the plugins repository
git clone https://github.com/lightningd/plugins
# Make sure the helpme plugin is executable (git should have already handled this)
chmod +x plugins/helpme/helpme.py
# Install its dependencies (there is only one actually)
pip3 install --user -r plugins/helpme/requirements.txt
# Then just start it :)
lightning-cli plugin start $PWD/plugins/helpme/helpme.py
```

The plugin registers a new command `helpme` which will guide you through the main components of C-lightning:

```
lightning-cli helpme
```

3.2.2 How to get the balance of each channel ?

You can use the `listfunds` command and take a ratio of `our_amount_msat` over `amount_msat`. Note that this doesn't account for the channel reserve.

A better option is to use the `summary` plugin which nicely displays channel balances, along with other useful channel information.

3.2.3 My channel is in state `STATE`, what does that mean ?

See the listpeers command manpage.

3.2.4 My payment is failing / all my payments are failing, why ?

There are many reasons for a payment failure. The most common one is a failure along the route from you to the payee. The best (and most common) solution to a route failure problem is to open more channels, which should increase the available routes to the recipient and lower the probability of a failure.

Hint: use the *pay* command which is will iterate through trying all possible routes, instead of the low-level `sendpay` command which only tries the passed in route.

3.2.5 How can I receive payments ?

In order to receive payments you need inbound liquidity. You get inbound liquidity when another node opens a channel to you or by successfully completing a payment out through a channel you opened.

If you need a lot of inbound liquidity, you can use a service that trustlessly swaps on-chain Bitcoin for Lightning channel capacity. There are a few online service providers that will create channels to you. A few of them charge fees for this service. Note that if you already have a channel open to them, you'll need to close it before requesting another channel.

3.2.6 Are there any issues if my node changes its IP address? What happens to the channels if it does?

There is no risk to your channels if your IP address changes. Other nodes might not be able to connect to you, but your node can still connect to them. But Core Lightning also has an integrated IPv4/6 address discovery mechanism. If your node detects an new public address, it will update its announcement. For this to work binhind a NAT router you need to forward the default TCP port 9735 to your node. IP discovery is only active if no other addresses are announced.

Alternatively, you can *setup a TOR hidden service* for your node that will also work well behind NAT firewalls.

3.2.7 Can I have two hosts with the same public key and different IP addresses, both online and operating at the same time?

No.

3.2.8 Can I use a single `bitcoind` for multiple `lightningd` ?

Yes. All `bitcoind` calls are handled by the bundled `bcli` plugin. `lightningd` does not use `bitcoind`'s wallet. While on the topic, `lightningd` does not require the `-txindex` option on `bitcoind`.

If you use a single `bitcoind` for multiple `lightningd`'s, be sure to raise the `bitcoind` max RPC thread limit (`-rpcthreads`), each `lightningd` can use up to 4 threads, which is the default `bitcoind` max.

3.2.9 Can I use Core Lightning on mobile ?

Remote control

Spark-wallet is the most popular remote control HTTP server for `lightningd`. **Use it behind tor**.

`lightningd` on Android

Effort has been made to get `lightningd` running on Android, see issue #3484. Currently unusable.

3.2.10 How to "backup my wallet" ?

See BACKUP.md for a more comprehensive discussion of your options.

In summary: as a Bitcoin user, one may be familiar with a file or a seed (or some mnemonics) from which it can recover all its funds.

Core Lightning has an internal bitcoin wallet, which you can use to make "on-chain" transactions, (see withdraw). These on-chain funds are backed up via the HD wallet seed, stored in byte-form in `hsm_secret`.

`lightningd` also stores information for funds locked in Lightning Network channels, which are stored in a database. This database is required for on-going channel updates as well as channel closure. There is no single-seed backup for funds locked in channels.

While crucial for node operation, snapshot-style backups of the `lightningd` database is **discouraged**, as *any* loss of state may result in permanent loss of funds. See the penalty mechanism for more information on why any amount of state-loss results in fund loss.

Real-time database replication is the recommended approach to backing up node data. Tools for replication are currently in active development, using the `db_write` plugin hook.

3.3 Channel Management

3.3.1 How to forget about a channel?

Channels may end up stuck during funding and never confirm on-chain. There is a variety of causes, the most common ones being that the funds have been double-spent, or the funding fee was too low to be confirmed. This is unlikely to happen in normal operation, as CLN tries to use sane defaults and prevents double-spends whenever possible, but using custom feerates or when the bitcoin backend has no good fee estimates it is still possible.

Before forgetting about a channel it is important to ensure that the funding transaction will never be confirmable by double-spending the funds. To do so you have to rescan the UTXOs using *`dev-rescan-outputs`* to reset any funds that may have been used in the funding transaction, then move all the funds to a new address:

```
lightning-cli dev-rescan-outputs
ADDR=$(lightning-cli newaddr bech32 | jq .bech32)
lightning-cli withdraw $ADDR all
```

This step is not required if the funding transaction was already double-spent, however it is safe to do it anyway, just in case.

Then wait for the transaction moving the funds to confirm. This ensures any pending funding transaction can no longer be confirmed.

As an additional step you can also force-close the unconfirmed channel:

```
lightning-cli close $PEERID 10  # Force close after 10 seconds
```

This will store a unilateral close TX in the DB as last resort means of recovery should the channel unexpectedly confirm anyway.

Now you can use the `dev-forget-channel` command to remove the DB entries from the database.

```
lightning-cli dev-forget-channel $NODEID
```

This will perform additional checks on whether it is safe to forget the channel, and only then removes the channel from the DB. Notice that this command is only available if CLN was compiled with `DEVELOPER=1`.

3.3.2 My channel is stuck in state `CHANNELD_AWAITING_LOCKIN`

There are two root causes to this issue:

- Funding transaction isn't confirmed yet. In this case we have to wait longer, or, in the case of a transaction that'll never confirm, forget the channel safely.
- The peer hasn't sent a lockin message. This message ackowledges that the node has seen sufficiently many confirmations to consider the channel funded.

In the case of a confirmed funding transaction but a missing lockin message, a simple reconnection may be sufficient to nudge it to acknowledge the confirmation:

```
lightning-cli disconnect $PEERID true  # force a disconnect
lightning-cli connect $PEERID
```

The lack of funding locked messages is a bug we are trying to debug here at issue #5366, if you have encountered this issue please drop us a comment and any information that may be helpful.

If this didn't work it could be that the peer is simply not caught up with the blockchain and hasn't seen the funding confirm yet. In this case we can either wait or force a unilateral close:

```
lightning-cli close $PEERID 10  # Force a unilateral after 10 seconds
```

If the funding transaction is not confirmed we may either wait or attempt to double-spend it. Confirmations may take a long time, especially when the fees used for the funding transaction were low. You can check if the transaction is still going to confirm by looking the funding transaction on a block explorer:

```
TXID=$(lightning-cli listpeers $PEERID | jq -r ''.peers[].channels[].funding_txid')
```

This will give you the funding transaction ID that can be looked up in any explorer.

If you don't want to wait for the channel to confirm, you could forget the channel (see *How to forget about a channel?* for details), however be careful as that may be dangerous and you'll need to rescan and double-spend the outputs so the funding cannot confirm.

3.4 Loss of funds

3.4.1 Rescanning the block chain for lost utxos

There are 3 types of 'rescans' you can make:

- `rescanblockchain`: A `bitcoind` RPC call which rescans the blockchain starting at the given height. This does not have an effect on Core Lightning as `lightningd` tracks all block and wallet data independently.
- `--rescan=depth`: A `lightningd` configuration flag. This flag is read at node startup and tells lightningd at what depth from current blockheight to rebuild its internal state. (You can specify an exact block to start scanning from, instead of depth from current height, by using a negative number.)
- `dev-rescan-outputs`: A `lightningd` RPC call. Only available if your node has been configured and built in DEVELOPER mode (i.e. `./configure --enable-developer`) This will sync the state for known UTXOs in the `lightningd` wallet with `bitcoind`. As it only operates on outputs already seen on chain by the `lightningd` internal wallet, this will not find missing wallet funds.

3.4.2 Database corruption / channel state lost

If you lose data (likely corrupted `lightningd.sqlite3`) about a channel **with `option_static_remotekey` enabled**, you can wait for your peer to unilateraly close the channel, then use `tools/hsmtool` with the `guesstoremote` command to attempt to recover your funds from the peer's published unilateral close transaction.

If `option_static_remotekey` was not enabled, you're probably out of luck. The keys for your funds in your peer's unilateral close transaction are derived from information you lost. Fortunately, since version `0.7.3` channels are created with `option_static_remotekey` by default if your peer supports it. Which is to say that channels created after block 598000 (short channel id starting with > 598000) have a high chance of supporting `option_static_remotekey`.

You can verify it using the `features` field from the `listpeers` command's result.

Here is an example in Python checking if one of the `option_static_remotekey` bits is set in the negotiated features corresponding to `0x02aaa2`:

```
>>> bool(0x02aaa2 & ((1 << 12) | (1 << 13)))
True
```

If `option_static_remotekey` is enabled you can attempt to recover the funds in a channel following this tutorial on how to extract the necessary information from the network topology. If successful, result will be a private key matching a unilaterally closed channel, that you can import into any wallet, recovering the funds into that wallet.

3.5 Technical Questions

3.5.1 How do I get the `psbt` for RPC calls that need it?

A `psbt` is created and returned by a call to `utxopsbt` with `reservedok=true`.

CHAPTER 4

Backing Up Your C-Lightning Node

Lightning Network channels get their scalability and privacy benefits from the very simple technique of *not telling anyone else about your in-channel activity*. This is in contrast to onchain payments, where you have to tell everyone about each and every payment and have it recorded on the blockchain, leading to scaling problems (you have to push data to everyone, everyone needs to validate every transaction) and privacy problems (everyone knows every payment you were ever involved in).

Unfortunately, this removes a property that onchain users are so used to, they react in surprise when learning about this removal. Your onchain activity is recorded in all archival fullnodes, so if you forget all your onchain activity because your storage got fried, you just go redownload the activity from the nearest archival fullnode.

But in Lightning, since *you* are the only one storing all your financial information, you ***cannot*** recover this financial information from anywhere else.

This means that on Lightning, **you have to** responsibly back up your financial information yourself, using various processes and automation.

The discussion below assumes that you know where you put your `$LIGHTNINGDIR`, and you know the directory structure within. By default your `$LIGHTNINGDIR` will be in `~/.lightning/${COIN}`. For example, if you are running `--mainnet`, it will be `~/.lightning/bitcoin`.

4.1 `hsm_secret`

`/!\` WHO SHOULD DO THIS: Everyone.

You need a copy of the `hsm_secret` file regardless of whatever backup strategy you use.

The `hsm_secret` is created when you first create the node, and does not change. Thus, a one-time backup of `hsm_secret` is sufficient.

This is just 32 bytes, and you can do something like the below and write the hexadecimal digits a few times on a piece of paper:

```
cd $LIGHTNINGDIR
xxd hsm_secret
```

You can re-enter the hexdump into a text file later and use `xxd` to convert it back to a binary `hsm_secret`:

```
cat > hsm_secret_hex.txt <<HEX
00: 30cc f221 94e1 7f01 cd54 d68c a1ba f124
10: e1f3 1d45 d904 823c 77b7 1e18 fd93 1676
HEX
xxd -r hsm_secret_hex.txt > hsm_secret
chmod 0400 hsm_secret
```

Notice that you need to ensure that the `hsm_secret` is only readable by the user, and is not writable, as otherwise `lightningd` will refuse to start. Hence the `chmod 0400 hsm_secret` command.

Alternatively, if you are deploying a new node that has no funds and channels yet, you can generate BIP39 words using any process, and create the `hsm_secret` using the `hsmtool generatehsm` command. If you did `make install` then `hsmtool` is installed as `lightning-hsmtool`, else you can find it in the `tools/` directory of the build directory.

```
lightning-hsmtool generatehsm hsm_secret
```

Then enter the BIP39 words, plus an optional passphrase. Then copy the `hsm_secret` to `${LIGHTNINGDIR}`

You can regenerate the same `hsm_secret` file using the same BIP39 words, which again, you can back up on paper.

Recovery of the `hsm_secret` is sufficient to recover any onchain funds. Recovery of the `hsm_secret` is necessary, but insufficient, to recover any in-channel funds. To recover in-channel funds, you need to use one or more of the other backup strategies below.

4.2 SQLITE3 `--wallet=${main}:${backup}` And Remote NFS Mount

`/!\` WHO SHOULD DO THIS: Casual users.

`/!\` **CAUTION** `/!\` This technique is only supported after the version v0.10.2 (not included) or later. On earlier versions, the `:` character is not special and will be considered part of the path of the database file.

When using the SQLITE3 backend (the default), you can specify a second database file to replicate to, by separating the second file with a single `:` character in the `--wallet` option, after the main database filename.

For example, if the user running `lightningd` is named `user`, and you are on the Bitcoin mainnet with the default `${LIGHTNINGDIR}`, you can specify in your `config` file:

```
wallet=sqlite3:///home/user/.lightning/bitcoin/lightningd.sqlite3:/my/backup/
↪lightningd.sqlite3
```

Or via command line:

```
lightningd --wallet=sqlite3:///home/user/.lightning/bitcoin/lightningd.sqlite3:/my/
↪backup/lightningd.sqlite3
```

If the second database file does not exist but the directory that would contain it does exist, the file is created. If the directory of the second database file does not exist, `lightningd` will fail at startup. If the second database file already exists, on startup it will be overwritten with the main database. During operation, all database updates will be done on both databases.

The main and backup files will **not** be identical at every byte, but they will still contain the same data.

It is recommended that you use **the same filename** for both files, just on different directories.

This has the advantage compared to the `backup` plugin below of requiring exactly the same amount of space on both the main and backup storage. The `backup` plugin will take more space on the backup than on the main storage. It has the disadvantage that it will only work with the SQLITE3 backend and is not supported by the PostgreSQL backend, and is unlikely to be supported on any future database backends.

You can only specify *one* replica.

It is recommended that you use a network-mounted filesystem for the backup destination. For example, if you have a NAS you can access remotely.

At the minimum, set the backup to a different storage device. This is no better than just using RAID-1 (and the RAID-1 will probably be faster) but this is easier to set up — just plug in a commodity USB flash disk (with metal casing, since a lot of writes are done and you need to dissipate the heat quickly) and use it as the backup location, without repartitioning your OS disk, for example.

Do note that files are not stored encrypted, so you should really not do this with rented space ("cloud storage").

To recover, simply get **all** the backup database files. Note that SQLITE3 will sometimes create a `-journal` or `-wal` file, which is necessary to ensure correct recovery of the backup; you need to copy those too, with corresponding renames if you use a different filename for the backup database, e.g. if you named the backup `backup.sqlite3` and when you recover you find `backup.sqlite3` and `backup.sqlite3-journal` files, you rename `backup.sqlite3` to `lightningd.sqlite3` and `backup.sqlite3-journal` to `lightningd.sqlite3-journal`. Note that the `-journal` or `-wal` file may or may not exist, but if they *do*, you *must* recover them as well (there can be an `-shm` file as well in WAL mode, but it is unnecessary; it is only used by SQLITE3 as a hack for portable shared memory, and contains no useful data; SQLITE3 will ignore its contents always). It is recommended that you use **the same filename** for both main and backup databases (just on different directories), and put the backup in its own directory, so that you can just recover all the files in that directory without worrying about missing any needed files or correctly renaming.

If your backup destination is a network-mounted filesystem that is in a remote location, then even loss of all hardware in one location will allow you to still recover your Lightning funds.

However, if instead you are just replicating the database on another storage device in a single location, you remain vulnerable to disasters like fire or computer confiscation.

4.3 `backup` Plugin And Remote NFS Mount

`/!\` WHO SHOULD DO THIS: Casual users.

You can find the full source for the `backup` plugin here: https://github.com/lightningd/plugins/tree/master/backup

The `backup` plugin requires Python 3.

- Download the source for the plugin.
  - `git clone https://github.com/lightningd/plugins.git`
- `cd` into its directory and install requirements.
  - `cd plugins/backup`
  - `pip3 install -r requirements.txt`
- Figure out where you will put the backup files.
  - Ideally you have an NFS or other network-based mount on your system, into which you will put the backup.
- Stop your Lightning node.

- `/path/to/backup-cli init --lightning-dir ${LIGHTNINGDIR} file:///path/to/nfs/mount/file.bkp`. This creates an initial copy of the database at the NFS mount.
- Add these settings to your `lightningd` configuration:
  - `important-plugin=/path/to/backup.py`
- Restart your Lightning node.

It is recommended that you use a network-mounted filesystem for the backup destination. For example, if you have a NAS you can access remotely.

Do note that files are not stored encrypted, so you should really not do this with rented space ("cloud storage").

Alternately, you *could* put it in another storage device (e.g. USB flash disk) in the same physical location.

To recover:

- Re-download the `backup` plugin and install Python 3 and the requirements of `backup`.
- `/path/to/backup-cli restore file:///path/to/nfs/mount ${LIGHTNINGDIR}`

If your backup destination is a network-mounted filesystem that is in a remote location, then even loss of all hardware in one location will allow you to still recover your Lightning funds.

However, if instead you are just replicating the database on another storage device in a single location, you remain vulnerable to disasters like fire or computer confiscation.

4.4 Filesystem Redundancy

`/!\` WHO SHOULD DO THIS: Filesystem nerds, data hoarders, home labs, enterprise users.

You can set up a RAID-1 with multiple storage devices, and point the `$LIGHTNINGDIR` to the RAID-1 setup. That way, failure of one storage device will still let you recover funds.

You can use a hardware RAID-1 setup, or just buy multiple commodity storage media you can add to your machine and use a software RAID, such as (not an exhaustive list!):

- `mdadm` to create a virtual volume which is the RAID combination of multiple physical media.
- BTRFS RAID-1 or RAID-10, a filesystem built into Linux.
- ZFS RAID-Z, a filesystem that cannot be legally distributed with the Linux kernel, but can be distributed in a BSD system, and can be installed on Linux with some extra effort, see ZFSonLinux.

RAID-1 (whether by hardware, or software) like the above protects against failure of a single storage device, but does not protect you in case of certain disasters, such as fire or computer confiscation.

You can "just" use a pair of high-quality metal-casing USB flash devices (you need metal-casing since the devices will have a lot of small writes, which will cause a lot of heating, which needs to dissipate very fast, otherwise the flash device firmware will internally disconnect the flash device from your computer, reducing your reliability) in RAID-1, if you have enough USB ports.

4.4.1 Example: BTRFS on Linux

On a Linux system, one of the simpler things you can do would be to use BTRFS RAID-1 setup between a partition on your primary storage and a USB flash disk. The below "should" work, but assumes you are comfortable with low-level Linux administration. If you are on a system that would make you cry if you break it, you **MUST** stop your Lightning node and back up all files before doing the below.

- Install `btrfs-progs` or `btrfs-tools` or equivalent.

- Get a 32Gb USB flash disk.
- Stop your Lightning node and back up everything, do not be stupid.
- Repartition your hard disk to have a 30Gb partition.
  - This is risky and may lose your data, so this is best done with a brand-new hard disk that contains no data.
- Connect the USB flash disk.
- Find the `/dev/sdXX` devices for the HDD 30Gb partition and the flash disk.
  - `lsblk -o NAME,TYPE,SIZE,MODEL` should help.
- Create a RAID-1 `btrfs` filesystem.
  - `mkfs.btrfs -m raid1 -d raid1 /dev/${HDD30GB} /dev/${USB32GB}`
  - You may need to add `-f` if the USB flash disk is already formatted.
- Create a mountpoint for the `btrfs` filesystem.
- Create a `/etc/fstab` entry.
  - Use the `UUID` option instad of `/dev/sdXX` since the exact device letter can change across boots.
  - You can get the UUID by `lsblk -o NAME,UUID`. Specifying *either* of the devices is sufficient.
  - Add `autodefrag` option, which tends to work better with SQLITE3 databases.
  - e.g. `UUID=${UUID} ${BTRFSMOUNTPOINT} btrfs defaults,autodefrag 0 0`
- `mount -a` then `df` to confirm it got mounted.
- Copy the contents of the `$LIGHTNINGDIR` to the BTRFS mount point.
  - Copy the entire directory, then `chown -R` the copy to the user who will run the `lightningd`.
  - If you are paranoid, run `diff -r` on both copies to check.
- Remove the existing `$LIGHTNINGDIR`.
- `ln -s ${BTRFSMOUNTPOINT}/lightningdirname ${LIGHTNINGDIR}`.
  - Make sure the `$LIGHTNINGDIR` has the same structure as what you originally had.
- Add `crontab` entries for `root` that perform regular `btrfs` maintenance tasks.
  - `0 0 * * * /usr/bin/btrfs balance start -dusage=50 -dlimit=2 -musage=50 -mlimit=4 ${BTRFSMOUNTPOINT}` This prevents BTRFS from running out of blocks even if it has unused space *within* blocks, and is run at midnight everyday. You may need to change the path to the `btrfs` binary.
  - `0 0 * * 0 /usr/bin/btrfs scrub start -B -c 2 -n 4 ${BTRFSMOUNTPOINT}` This detects bit rot (i.e. bad sectors) and auto-heals the filesystem, and is run on Sundays at midnight.
- Restart your Lightning node.

If one or the other device fails completely, shut down your computer, boot on a recovery disk or similar, then:

- Connect the surviving device.
- Mount the partition/USB flash disk in `degraded` mode:
  - `mount -o degraded /dev/sdXX /mnt/point`
- Copy the `lightningd.sqlite3` and `hsm_secret` to new media.
  - Do **not** write to the degraded `btrfs` mount!

- Start up a `lightningd` using the `hsm_secret` and `lightningd.sqlite3` and close all channels and move all funds to onchain cold storage you control, then set up a new Lightning node.

If the device that fails is the USB flash disk, you can replace it using BTRFS commands. You should probably stop your Lightning node while doing this.

- `btrfs replace start /dev/sdOLD /dev/sdNEW ${BTRFSMOUNTPOINT}`.
  - If `/dev/sdOLD` no longer even exists because the device is really really broken, use `btrfs filesystem show` to see the number after `devid` of the broken device, and use that number instead of `/dev/sdOLD`.
- Monitor status with `btrfs replace status ${BTRFSMOUNTPOINT}`.

More sophisticated setups with more than two devices are possible. Take note that "RAID 1" in `btrfs` means "data is copied on up to two devices", meaning only up to one device can fail. You may be interested in `raid1c3` and `raid1c4` modes if you have three or four storage devices. BTRFS would probably work better if you were purchasing an entire set of new storage devices to set up a new node.

4.5 PostgreSQL Cluster

`/!\` WHO SHOULD DO THIS: Enterprise users, whales.

`lightningd` may also be compiled with PostgreSQL support. PostgreSQL is generally faster than SQLITE3, and also supports running a PostgreSQL cluster to be used by `lightningd`, with automatic replication and failover in case an entire node of the PostgreSQL cluster fails.

Setting this up, however, is more involved.

By default, `lightningd` compiles with PostgreSQL support **only** if it finds `libpq` installed when you `./configure`. To enable it, you have to install a developer version of `libpq`. On most Debian-derived systems that would be `libpq-dev`. To verify you have it properly installed on your system, check if the following command gives you a path:

```
pg_config --includedir
```

Versioning may also matter to you. For example, Debian Stable ("buster") as of late 2020 provides PostgreSQL 11.9 for the `libpq-dev` package, but Ubuntu LTS ("focal") of 2020 provides PostgreSQL 12.5. Debian Testing ("bullseye") uses PostgreSQL 13.0 as of this writing. PostgreSQL 12 had a non-trivial change in the way the restore operation is done for replication. You should use the same PostgreSQL version of `libpq-dev` as what you run on your cluster, which probably means running the same distribution on your cluster.

Once you have decided on a specific version you will use throughout, refer as well to the "synchronous replication" document of PostgreSQL for the **specific version** you are using:

- PostgreSQL 11
- PostgreSQL 12
- PostgreSQL 13

You then have to compile `lightningd` with PostgreSQL support.

- Clone or untar a new source tree for `lightning` and `cd` into it.
  - You *could* just use `make clean` on an existing one, but for the avoidance of doubt (and potential bugs in our `Makefile` cleanup rules), just create a fresh source tree.
- `./configure`
  - Add any options to `configure` that you normally use as well.

- Double-check the `config.vars` file contains `HAVE_POSTGRES=1`.
  - `grep 'HAVE_POSTGRES' config.vars`
- `make`
- If you install `lightningd`, `sudo make install`.

If you were not using PostgreSQL before but have compiled and used `lightningd` on your system, the resulting `lightningd` will still continue supporting and using your current SQLITE3 database; it just gains the option to use a PostgreSQL database as well.

If you just want to use PostgreSQL without using a cluster (for example, as an initial test without risking any significant funds), then after setting up a PostgreSQL database, you just need to add `--wallet=postgres://${USER}:${PASSWORD}@${HOST}:${PORT}/${DB}` to your `lightningd` config or invocation.

To set up a cluster for a brand new node, follow this (external) guide by @gabridome.

The above guide assumes you are setting up a new node from scratch. It is also specific to PostgreSQL 12, and setting up for other versions **will** have differences; read the PostgreSQL manuals linked above.

If you want to continue a node that started using an SQLITE3 database, note that we do not support this. You should set up a new PostgreSQL node, move funds from the SQLITE3 node to the PostgreSQL node, then shut down the SQLITE3 node permanently.

There are also more ways to set up PostgreSQL replication. In general, you should use synchronous replication (13), since `lightningd` assumes that once a transaction is committed, it is saved in all permanent storage. This can be difficult to create remote replicas due to the latency.

4.6 SQLite Litestream Replication

`/!\` **CAUTION** `/!\` Previous versions of this document recommended this technique, but we no longer do so. According to issue 4857, even with a 60-second timeout that we added in 0.10.2, this leads to constant crashing of `lightningd` in some situations. This section will be removed completely six months after 0.10.3. Consider using `--wallet=sqlite3://${main}:${backup}` above, instead.

One of the simpler things on any system is to use Litestream to replicate the SQLite database. It continuously streams SQLite changes to file or external storage - the cloud storage option should not be used. Backups/replication should not be on the same disk as the original SQLite DB.

You need to enable WAL mode on your database. To do so, first stop `lightningd`, then:

```
$ sqlite3 lightningd.sqlite3
sqlite3> PRAGMA journal_mode = WAL;
sqlite3> .quit
```

Then just restart `lightningd`.

/etc/litestream.yml :

```
dbs:
 - path: /home/bitcoin/.lightning/bitcoin/lightningd.sqlite3
   replicas:
     - path: /media/storage/lightning_backup
```

and start the service using systemctl:

```
$ sudo systemctl start litestream
```

Restore:

```
$ litestream restore -o /media/storage/lightning_backup  /home/bitcoin/restore_
↪lightningd.sqlite3
```

Because Litestream only copies small changes and not the entire database (holding a read lock on the file while doing so), the 60-second timeout on locking should not be reached unless something has made your backup medium very very slow.

Litestream has its own timer, so there is a tiny (but non-negligible) probability that `lightningd` updates the database, then irrevocably commits to the update by sending revocation keys to the counterparty, and *then* your main storage media crashes before Litestream can replicate the update. Treat this as a superior version of "Database File Backups" section below and prefer recovering via other backup methods first.

4.7 Database File Backups

`/!\` WHO SHOULD DO THIS: Those who already have at least one of the other backup methods, those who are #reckless.

This is the least desirable backup strategy, as it *can* lead to loss of all in-channel funds if you use it. However, having *no* backup strategy at all *will* lead to loss of all in-channel funds, so this is still better than nothing.

This backup method is undesirable, since it cannot recover the following channels:

- Channels with peers that do not support `option_dataloss_protect`.
  - Most nodes on the network already support `option_dataloss_protect` as of November 2020.
  - If the peer does not support `option_dataloss_protect`, then the entire channel funds will be revoked by the peer.
  - Peers can *claim* to honestly support this, but later steal funds from you by giving obsolete state when you recover.
- Channels created *after* the copy was made are not recoverable.
  - Data for those channels does not exist in the backup, so your node cannot recover them.

Because of the above, this strategy is discouraged: you *can* potentially lose all funds in open channels.

However, again, note that a "no backups #reckless" strategy leads to *definite* loss of funds, so you should still prefer *this* strategy rather than having *no* backups at all.

Even if you have one of the better options above, you might still want to do this as a worst-case fallback, as long as you:

- Attempt to recover using the other backup options above first. Any one of them will be better than this backup option.
- Recover by this method **ONLY** as a ***last*** resort.
- Recover using the most recent backup you can find. Take time to look for the most recent available backup.

Again, this strategy can lead to only ***partial*** recovery of funds, or even to complete failure to recover, so use the other methods first to recover!

4.7.1 Offline Backup

While `lightningd` is not running, just copy the `lightningd.sqlite3` file in the `$LIGHTNINGDIR` on backup media somewhere.

To recover, just copy the backed up `lightningd.sqlite3` into your new `$LIGHTNINGDIR` together with the `hsm_secret`.

You can also use any automated backup system as long as it includes the `lightningd.sqlite3` file (and optionally `hsm_secret`, but note that as a secret key, thieves getting a copy of your backups may allow them to steal your funds, even in-channel funds) and as long as it copies the file while `lightningd` is not running.

4.7.2 Backing Up While `lightningd` Is Running

Since `sqlite3` will be writing to the file while `lightningd` is running, `cp`ing the `lightningd.sqlite3` file while `lightningd` is running may result in the file not being copied properly if `sqlite3` happens to be committing database transactions at that time, potentially leading to a corrupted backup file that cannot be recovered from.

You have to stop `lightningd` before copying the database to backup in order to ensure that backup files are not corrupted, and in particular, wait for the `lightningd` process to exit. Obviously, this is disruptive to node operations, so you might prefer to just perform the `cp` even if the backup potentially is corrupted. As long as you maintain multiple backups sampled at different times, this may be more acceptable than stopping and restarting `lightningd`; the corruption only exists in the backup, not in the original file.

If the filesystem or volume manager containing `$LIGHTNINGDIR` has a snapshot facility, you can take a snapshot of the filesystem, then mount the snapshot, copy `lightningd.sqlite3`, unmount the snapshot, and then delete the snapshot. Similarly, if the filesystem supports a "reflink" feature, such as `cp -c` on an APFS on MacOS, or `cp --reflink=always` on an XFS or BTRFS on Linux, you can also use that, then copy the reflinked copy to a different storage medium; this is equivalent to a snapshot of a single file. This *reduces* but does not *eliminate* this race condition, so you should still maintain multiple backups.

You can additionally perform a check of the backup by this command:

```
echo 'PRAGMA integrity_check;' | sqlite3 ${BACKUPFILE}
```

This will result in the string `ok` being printed if the backup is **likely** not corrupted. If the result is anything else than `ok`, the backup is definitely corrupted and you should make another copy.

In order to make a proper uncorrupted backup of the SQLITE3 file while `lightningd` is running, we would need to have `lightningd` perform the backup itself, which, as of the version at the time of this writing, is not yet implemented.

Even if the backup is not corrupted, take note that this backup strategy should still be a last resort; recovery of all funds is still not assured with this backup strategy.

`sqlite3` has `.dump` and `VACUUM INTO` commands, but note that those lock the main database for long time periods, which will negatively affect your `lightningd` instance.

4.7.3 `sqlite3` `.dump` or `VACUUM INTO` Commands

`/!\` **CAUTION** `/!\` Previous versions of this document recommended this technique, but we no longer do so. According to issue 4857, even with a 60-second timeout that we added in 0.10.2, this may lead to constant crashing of `lightningd` in some situations; this technique uses substantially the same techniques as `litestream`. This section will be removed completely six months after 0.10.3. Consider using `--wallet=sqlite3://${main}:${backup}` above, instead.

Use the `sqlite3` command on the `lightningd.sqlite3` file, and feed it with `.dump "/path/to/backup.sqlite3"` or `VACUUM INTO "/path/to/backup.sqlite3";`.

These create a snapshot copy that, unlike the previous technique, is assuredly uncorrupted (barring any corruption caused by your backup media).

However, if the copying process takes a long time (approaching the timeout of 60 seconds) then you run the risk of `lightningd` attempting to grab a write lock, waiting up to 60 seconds, and then failing with a "database is locked" error. Your backup system could `.dump` to a fast `tmpfs` RAMDISK or local media, and *then* copy to the final backup media on a remote system accessed via slow network, for example, to reduce this risk.

It is recommended that you use `.dump` instead of `VACUUM INTO`, as that is assuredly faster; you can just open the backup copy in a new `sqlite3` session and `VACUUM;` to reduce the size of the backup.

CHAPTER 5

Plugins

Plugins are a simple yet powerful way to extend the functionality provided by Core 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[^options].
- **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`)

A `helloworld.py` example plugin based on pyln-client can be found here. There is also a repository with a collection of actively maintained plugins and finally, `lightningd`'s own internal tests can be a useful (and most reliable) resource.

[^options]: Only for plugins that start when `lightningd` starts, option values are not remembered when a plugin is stopped or killed.

5.1 Warning

As noted, `lightningd` uses `stdin` as an intake mechanism. This can cause unexpected behavior if one is not careful. To wit, care should be taken to ensure that debug/logging statements must be routed to `stderr` or directly to a file. Activities that are benign in other contexts (`println!`, `dbg!`, etc) will cause the plugin to be killed with an error along the lines of:

```
UNUSUAL plugin-cln-plugin-startup: Killing plugin: JSON-RPC message does not
contain "jsonrpc" field
```

5.2 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 `~/.lightning/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 as working directory and env variables "LIGHTNINGD_PLUGIN" and "LIGHTNINGD_VERSION" set, 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.

Above is generally valid for plugins that start when `lightningd` starts. For dynamic plugins that start via the *plugin* JSON-RPC command there is some difference, mainly in options passthrough (see note in *Types of Options*).

- `shutdown` (optional): if subscribed to "shutdown" notification, a plugin can exit cleanly when `lightningd` is shutting down or when stopped via `plugin stop`.

5.2.1 The `getmanifest` method

The `getmanifest` method is required for all plugins and will be called on startup with optional parameters (in particular, it may have `allow-deprecated-apis: false`, but you should accept, and ignore, other parameters). It MUST return a JSON object similar to this example:

```
{
  "options": [
    {
      "name": "greeting",
      "type": "string",
      "default": "World",
      "description": "What name should I call you?",
      "deprecated": false
    }
  ],
  "rpcmethods": [
    {
      "name": "hello",
      "usage": "[name]",
      "description": "Returns a personalized greeting for {greeting} (set via␣
↪options)."
    },
    {
      "name": "gettime",
      "usage": "",
```

(continues on next page)

(continued from previous page)

```
      "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.",
      "deprecated": false
    }
  ],
  "subscriptions": [
    "connect",
    "disconnect"
  ],
  "hooks": [
    { "name": "openchannel", "before": ["another_plugin"] },
    { "name": "htlc_accepted" }
  ],
  "featurebits": {
    "node": "D0000000",
    "channel": "D0000000",
    "init": "0E000000",
    "invoice": "00AD0000"
  },
  "notifications": [
    {
        "method": "mycustomnotification"
      }
  ],
  "dynamic": true
}
```

During startup the `options` will be added to the list of command line options that `lightningd` accepts. If any `options` “name” is already taken startup will abort. The above will add a `--greeting` option with a default value of `World` and the specified description. *Notice that currently string, integers, bool, and flag 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 `[]`.

`options` and `rpcmethods` can mark themselves `deprecated: true` if you plan on removing them: this will disable them if the user sets `allow-deprecated-apis` to false (which every developer should do, right?).

The `dynamic` indicates if the plugin can be managed after `lightningd` has been started using the *plugin* JSON-RPC command. Critical plugins that should not be stopped should set it to false. Plugin `options` can be passed to dynamic plugins as argument to the `plugin` command .

If a `disable` member exists, the plugin will be disabled and the contents of this member is the reason why. This allows plugins to disable themselves if they are not supported in this configuration.

The `featurebits` object allows the plugin to register featurebits that should be announced in a number of places in the protocol. They can be used to signal support for custom protocol extensions to direct peers, remote nodes and in invoices. Custom protocol extensions can be implemented for example using the `sendcustommsg` method and the `custommsg` hook, or the `sendonion` method and the `htlc_accepted` hook. The keys in the `featurebits` object are `node` for features that should be announced via the `node_announcement` to all nodes in the network, `init` for features that should be announced to direct peers during the connection setup, `channel` for features which should apply to `channel_announcement`, and `invoice` for features that should be announced to a potential sender of a payment in the invoice. The low range of featurebits is reserved for standardize features, so please pick random, high position bits for experiments. If you’d like to standardize your extension please reach out to the *specifi-*

cation repository to get a featurebit assigned.

The `notifications` array allows plugins to announce which custom notifications they intend to send to `lightningd`. These custom notifications can then be subscribed to by other plugins, allowing them to communicate with each other via the existing publish-subscribe mechanism and react to events that happen in other plugins, or collect information based on the notification topics.

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 kill the plugin, this aborts startup if the plugin is *important*.

Types of Options

There are currently four supported option 'types':

- string: a string
- bool: a boolean
- int: parsed as a signed integer (64-bit)
- flag: no-arg flag option. Is boolean under the hood. Defaults to false.

In addition, string and int types can specify `"multi": true` to indicate they can be specified multiple times. These will always be represented in `init` as a (possibly empty) JSON array.

Nota bene: if a `flag` type option is not set, it will not appear in the options set that is passed to the plugin.

Here's an example option set, as sent in response to `getmanifest`

```
"options": [
  {
    "name": "greeting",
    "type": "string",
    "default": "World",
    "description": "What name should I call you?"
  },
  {
    "name": "run-hot",
    "type": "flag",
    "default": None,  // defaults to false
    "description": "If set, overclocks plugin"
  },
  {
    "name": "is_online",
    "type": "bool",
    "default": false,
    "description": "Set to true if plugin can use network"
  },
  {
    "name": "service-port",
    "type": "int",
    "default": 6666,
    "description": "Port to use to connect to 3rd-party service"
  },
  {
    "name": "number",
    "type": "int",
    "default": 0,
    "description": "Another number to add",
```

(continues on next page)

(continued from previous page)

```
          "multi": true
    }
  ],
```

Note: `lightningd` command line options are only parsed during startup and their values are not remembered when the plugin is stopped or killed. For dynamic plugins started with `plugin start`, options can be passed as extra arguments to that *command*.

Custom notifications

The plugins may emit custom notifications for topics they have announced during startup. The list of notification topics declared during startup must include all topics that may be emitted, in order to verify that all topics plugins subscribe to are also emitted by some other plugin, and warn if a plugin subscribes to a non-existent topic. In case a plugin emits notifications it has not announced the notification will be ignored and not forwarded to subscribers.

When forwarding a custom notification `lightningd` will wrap the payload of the notification in an object that contains metadata about the notification. The following is an example of this transformation. The first listing is the original notification emitted by the `sender` plugin, while the second is the the notification as received by the `receiver` plugin (both listings show the full JSON-RPC notification to illustrate the wrapping).

```
{
  "jsonrpc": "2.0",
  "method": "mycustomnotification",
  "params": {
    "key": "value",
        "message": "Hello fellow plugin!"
  }
}
```

is delivered as

```
{
  "jsonrpc": "2.0",
  "method": "mycustomnotification",
  "params": {
    "origin": "sender",
    "payload": {
      "key": "value",
      "message": "Hello fellow plugin!"
    }
  }
}
```

The notification topic (`method` in the JSON-RPC message) must not match one of the internal events in order to prevent breaking subscribers that expect the existing notification format. Multiple plugins are allowed to emit notifications for the same topics, allowing things like metric aggregators where the aggregator subscribes to a common topic and other plugins publish metrics as notifications.

5.2.2 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",
        "number": [0]
  },
  "configuration": {
    "lightning-dir": "/home/user/.lightning/testnet",
    "rpc-file": "lightning-rpc",
    "startup": true,
    "network": "testnet",
    "feature_set": {
        "init": "02aaa2",
        "node": "8000000002aaa2",
        "channel": "",
        "invoice": "028200"
    },
    "proxy": {
        "type": "ipv4",
        "address": "127.0.0.1",
        "port": 9050
    },
    "torv3-enabled": true,
    "always_use_proxy": false
  }
}
```

The plugin must respond to `init` calls. The response should be a valid JSON-RPC response to the `init`, but this is not currently enforced. If the response is an object containing `result` which contains `disable` then the plugin will be disabled and the contents of this member is the reason why.

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

5.2.3 Timeouts

During startup ("startup" is true), the plugin has 60 seconds to return `getmanifest` and another 60 seconds to return `init`, or gets killed. When started dynamically via the *plugin* JSON-RPC command, both `getmanifest` and `init` should be completed within 60 seconds.

5.3 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": [
    {
```

(continues on next page)

(continued from previous page)

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

5.4 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.

5.4.1 `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_msat": 100000000,
    "funding_txid": "4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b
↪",
```

(continues on next page)

(continued from previous page)

```
    "channel_ready": false
  }
}
```

5.4.2 channel_open_failed

A notification to indicate that a channel open attempt has been unsuccessful. Useful for cleaning up state for a v2 channel open attempt. See `plugins/funder.c` for an example of how to use this.

```
{
  "channel_open_failed": {
    "channel_id": "a2d0851832f0e30a0cf...",
  }
}
```

5.4.3 channel_state_changed

A notification for topic `channel_state_changed` is sent every time a channel changes its state. The notification includes the `peer_id` and `channel_id`, the old and new channel states, the type of `cause` and a `message`.

```
{
    "channel_state_changed": {
        "peer_id": "03bc9337c7a28bb784d67742ebedd30a93bacdf7e4ca16436ef3798000242b2251
↪",
        "channel_id":
↪"a2d0851832f0e30a0cf778a826d72f077ca86b69f72677e0267f23f63a0599b4",
        "short_channel_id" : "561820x1020x1",
        "old_state": "CHANNELD_NORMAL",
        "new_state": "CHANNELD_SHUTTING_DOWN",
        "cause" : "remote",
        "message" : "Peer closes channel"
    }
}
```

A `cause` can have the following values:

- “unknown” Anything other than the reasons below. Should not happen.
- “local” Unconscious internal reasons, e.g. dev fail of a channel.
- “user” The operator or a plugin opened or closed a channel by intention.
- “remote” The remote closed or funded a channel with us by intention.
- “protocol” We need to close a channel because of bad signatures and such.
- “onchain” A channel was closed onchain, while we were offline.

Most state changes are caused subsequentially for a prior state change, e.g. “CLOSINGD_COMPLETE” is followed by “FUNDING_SPEND_SEEN”. Because of this, the `cause` reflects the last known reason in terms of local or remote user interaction, protocol reasons, etc. More specifically, a `new_state` “FUNDING_SPEND_SEEN” will likely *not* have “onchain” as a `cause` but some value such as “REMOTE” or “LOCAL” depending on who initiated the closing of a channel.

Note: If the channel is not closed or being closed yet, the `cause` will reflect which side “remote” or “local” opened the channel.

Note: If the cause is "onchain" this was very likely a conscious decision of the remote peer, but we have been offline.

5.4.4 `connect`

A notification for topic `connect` is sent every time a new connection to a peer is established. `direction` is either `"in"` or `"out"`.

```json
{
  "id": "02f6725f9c1c40333b67faea92fd211c183050f28df32cac3f9d69685fe9665432",
  "direction": "in",
  "address": "1.2.3.4:1234"
}
```

5.4.5 `disconnect`

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

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

5.4.6 `invoice_payment`

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

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

5.4.7 `invoice_creation`

A notification for topic `invoice_creation` is sent every time an invoice is created.

```json
{
  "invoice_creation": {
    "label": "unique-label-for-invoice",
    "preimage": "0000000000000000000000000000000000000000000000000000000000000000",
    "amount_msat": 10000
  }
}
```

5.4.8 `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.

5.4.9 `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_msat": 100001001,
    "out_msat": 100000000,
    "fee_msat": 1001,
    "status": "settled",
    "received_time": 1560696342.368,
    "resolved_time": 1560696342.556
  }
}
```

or

```
{
  "forward_event": {
    "payment_hash": "ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
↪",
    "in_channel": "103x2x1",
    "out_channel": "110x1x0",
    "in_msat": 100001001,
    "out_msat": 100000000,
    "fee_msat": 1001,
    "status": "local_failed",
    "failcode": 16392,
    "failreason": "WIRE_PERMANENT_CHANNEL_FAILURE",
```

(continues on next page)

(continued from previous page)

```
    "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][bolt4-failure-codes].

5.4.10 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
↪",
    "amount_msat": 100000000,
    "amount_sent_msat": 100001001,
    "created_at": 1561390572,
    "status": "complete",
    "payment_preimage":
↪"9540d98095fd7f37687ebb7759e733934234d4f934e34433d4998a37de3733ee"
```

(continues on next page)

(continued from previous page)

```
  }
}
```

`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.

5.4.11 `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",
      "amount_msat": 100000000,
      "amount_sent_msat": 100001001,
      "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.

5.4.12 `coin_movement`

A notification for topic `coin_movement` is sent to record the movement of coins. It is only triggered by finalized ledger updates, i.e. only definitively resolved HTLCs or confirmed bitcoin transactions.

```
{
        "coin_movement": {
                "version":2,
                "node_id":
↪"03a7103a2322b811f7369cbb27fb213d30bbc0b012082fed3cad7e4498da2dc56b",
                "type":"chain_mvt",
                "account_id":"wallet",
                "originating_account": "wallet", // (`chain_mvt` only, optional)
                "txid":
↪"0159693d8f3876b4def468b208712c630309381e9d106a9836fa0a9571a28722", // (`chain_mvt`␣
↪only, optional)
```

(continues on next page)

(continued from previous page)

```
                "utxo_txid":
↪"0159693d8f3876b4def468b208712c630309381e9d106a9836fa0a9571a28722", // (`chain_mvt`␣
↪only)
                "vout":1, // (`chain_mvt` only)
                "payment_hash": "xxx", // (either type, optional on both)
                "part_id": 0, // (`channel_mvt` only, optional)
                "credit_msat":2000000000,
                "debit_msat":0,
                "output_msat": 2000000000, // ('chain_mvt' only)
                "output_count": 2, // ('chain_mvt' only, typically only channel␣
↪closes)
                "fees_msat": 382, // ('channel_mvt' only)
                "tags": ["deposit"],
                "blockheight":102, // 'chain_mvt' only
                "timestamp":1585948198,
                "coin_type":"bc"
        }
}
```

`version` indicates which version of the coin movement data struct this notification adheres to.

`node_id` specifies the node issuing the coin movement.

`type` marks the underlying mechanism which moved these coins. There are two 'types' of `coin_movements`:

- `channel_mvts`, which occur as a result of htlcs being resolved and,
- `chain_mvts`, which occur as a result of bitcoin txs being mined.

`account_id` is the name of this account. The node's wallet is named 'wallet', all channel funds' account are the channel id.

`originating_account` is the account that this movement originated from. *Only* tagged on external events (deposits/withdrawals to an external party).

`txid` is the transaction id of the bitcoin transaction that triggered this ledger event. `utxo_txid` and `vout` identify the bitcoin output which triggered this notification. (`chain_mvt` only). Notifications tagged `journal_entry` do not have a `utxo_txid` as they're not represented in the utxo set.

`payment_hash` is the hash of the preimage used to move this payment. Only present for HTLC mediated moves (both `chain_mvt` and `channel_mvt`) A `chain_mvt` will have a `payment_hash` iff it's recording an htlc that was fulfilled onchain.

`part_id` is an identifier for parts of a multi-part payment. useful for aggregating payments for an invoice or to indicate why a payment hash appears multiple times. `channel_mvt` only

`credit` and `debit` are millisatoshi denominated amounts of the fund movement. A 'credit' is funds deposited into an account; a `debit` is funds withdrawn.

`output_value` is the total value of the on-chain UTXO. Note that for channel opens/closes the total output value will not necessarily correspond to the amount that's credited/debited.

`output_count` is the total outputs to expect for a channel close. Useful for figuring out when every onchain output for a close has been resolved.

`fees` is an HTLC annotation for the amount of fees either paid or earned. For "invoice" tagged events, the fees are the total fees paid to send that payment. The end amount can be found by subtracting the total fees from the `debited` amount. For "routed" tagged events, both the debit/credit contain fees. Technically routed debits are the 'fee generating' event, however we include them on routed credits as well.

`tag` is a movement descriptor. Current tags are as follows:

- `deposit`: funds deposited
- `withdrawal`: funds withdrawn
- `penalty`: funds paid or gained from a penalty tx.
- `invoice`: funds paid to or recieved from an invoice.
- `routed`: funds routed through this node.
- `pushed`: funds pushed to peer.
- `channel_open` : channel is opened, initial channel balance
- `channel_close`: channel is closed, final channel balance
- `delayed_to_us`: on-chain output to us, spent back into our wallet
- `htlc_timeout`: on-chain htlc timeout output
- `htlc_fulfill`: on-chian htlc fulfill output
- `htlc_tx`: on-chain htlc tx has happened
- `to_wallet`: output being spent into our wallet
- `ignored`: output is being ignored
- `anchor`: an anchor output
- `to_them`: output intended to peer's wallet
- `penalized`: output we've 'lost' due to a penalty (failed cheat attempt)
- `stolen`: output we've 'lost' due to peer's cheat
- `to_miner`: output we've burned to miner (OP_RETURN)
- `opener`: tags channel_open, we are the channel opener
- `lease_fee`: amount paid as lease fee
- `leased`: tags channel_open, channel contains leased funds

`blockheight` is the block the txid is included in. `channel_mvts` will be null, so will the blockheight for withdrawals to external parties (we issue these events when we send the tx containing them, before they're included in the chain).

The `timestamp` is seconds since Unix epoch of the node's machine time at the time lightningd broadcasts the notification.

`coin_type` is the BIP173 name for the coin which moved.

5.4.13 `balance_snapshot`

Emitted after we've caught up to the chain head on first start. Lists all current accounts (`account_id` matches the `account_id` emitted from `coin_movement`). Useful for checkpointing account balances.

```
{
    "balance_snapshots": [
        {
            'node_id':
↪'035d2b1192dfba134e10e540875d366ebc8bc353d5aa766b80c090b39c3a5d885d',
            'blockheight': 101,
            'timestamp': 1639076327,
```

(continues on next page)

(continued from previous page)

```
            'accounts': [
                {
                    'account_id': 'wallet',
                    'balance': '0msat',
                    'coin_type': 'bcrt'
                }
            ]
        },
        {
            'node_id':
↪'035d2b1192dfba134e10e540875d366ebc8bc353d5aa766b80c090b39c3a5d885d',
            'blockheight': 110,
            'timestamp': 1639076343,
            'accounts': [
                {
                    'account_id': 'wallet',
                    'balance': '995433000msat',
                    'coin_type': 'bcrt'
                }, {
                    'account_id':
↪'5b65c199ee862f49758603a5a29081912c8816a7c0243d1667489d244d3d055f',
                     'balance': '500000000msat',
                    'coin_type': 'bcrt'
                }
            ]
        }
    ]
}
```

5.4.14 `block_added`

Emitted after each block is received from bitcoind, either during the initial sync or throughout the node's life as new blocks appear.

```
{
    "block": {
      "hash": "000000000000000000034bdb3c01652a0aa8f63d32f949313d55af2509f9d245",
      "height": 753304
    }
}
```

5.4.15 `openchannel_peer_sigs`

When opening a channel with a peer using the collaborative transaction protocol (`opt_dual_fund`), this notification is fired when the peer sends us their funding transaction signatures, `tx_signatures`. We update the in-progress PSBT and return it here, with the peer's signatures attached.

```
{
  "openchannel_peer_sigs": {
    "channel_id": "252d1b0a1e5789...",
    "signed_psbt": "cHNidP8BAKgCAAAAAQ+y+61AQAAAAD9////
↪AzbkHAAAAAAAFgAUwsyrFxwqW+natS7EG4JYYwJMVGZQwwAAAAAAACIAIKYE2s4YZ+RON6BB5lYQESHR9cA7hDm6/
↪maYtTzSLA0hUMMAAAAAAAAiACBbjNO5FM9nzdj6YnPJMDU902R2c0+9liECwt9TuQiAzWYAAAAAAQDfAgAAAAABARtaSZufCbC
↪G23XVaQ8mDwZQFW1vlCsCYhLbmVrpAAAAAAD+////
↪AvJs5ykBAAAAFgAUT6ORgb3CgFsbwSOzNLzF7jQS5s+AhB4AAAAAABepFNi369DMyAJmqX2agouvGHcDKsZkhwJHMEQCIHELIy
↪PDElnqWw49y2vTqauSJIVBgGtSc+vq5BQd+gEhAg0f8WITWvA8o4grxNKfgdrNDncqreMLeRFiteUlne+GZQAAAAEBIICEHgAAA
↪oF3SaF3czwu8CWxpZ2h0bmluZwEIexhVcpJl8ugM/
↪AlsaWdodG5pbmcCAgABAAz8CWxpZ2h0bmluZwEIR7FutlQgkSoADPwJbGlnaHRuaW5nAQhYT+HjxFBqeAAM/
↪AlsaWdodG5pbmcBCOpQ5iiTTNQEAA=="
```

(continues on next page)

(continued from previous page)

```
  }
}
```

5.4.16 `shutdown`

Send in two situations: lightningd is (almost completely) shutdown, or the plugin `stop` command has been called for this plugin. In both cases the plugin has 30 seconds to exit itself, otherwise it's killed.

In the shutdown case, plugins should not interact with lightnind except via (id-less) logging or notifications. New rpc calls will fail with error code -5 and (plugin's) responses will be ignored. Because lightningd can crash or be killed, a plugin cannot rely on the shutdown notification always been send.

5.5 Hooks

Hooks allow a plugin to define custom behavior for `lightningd` without having to modify the Core 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.

When hooks are registered, they can optionally specify "before" and "after" arrays of plugin names, which control what order they will be called in. If a plugin name is unknown, it is ignored, otherwise if the hook calls cannot be ordered to satisfy the specifications of all plugin hooks, the plugin registration will fail.

The call semantics of the hooks, i.e., when and how hooks are called, depend on the hook type. Most hooks are currently set to `single`-mode. In this mode only a single plugin can register the hook, and that plugin will get called for each event of that type. If a second plugin attempts to register the hook it gets killed and a corresponding log entry will be added to the logs.

In `chain`-mode multiple plugins can register for the hook type and they are called in any order they are loaded (i.e. cmdline order first, configuration order file second: though note that the order of plugin directories is implementation-dependent), overriden only by `before` and `after` requirements the plugin's hook registrations specify. Each plugin can then handle the event or defer by returning a `continue` result like the following:

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

The remainder of the response is ignored and if there are any more plugins that have registered the hook the next one gets called. If there are no more plugins then the internal handling is resumed as if no hook had been called. Any other result returned by a plugin is considered an exit from the chain. Upon exit no more plugin hooks are called for the current event, and the result is executed. Unless otherwise stated all hooks are `single`-mode.

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 and get notified in parallel, however only one plugin may register for `single`-mode hook types, and in all cases only one plugin may return a non-`continue` response. This avoids having multiple contradictory responses.

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.

As a convention, for all hooks, returning the object `{ "result" : "continue" }` results in `lightningd` behaving exactly as if no plugin is registered on the hook.

5.5.1 `peer_connected`

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

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

The hook is sparse on information, since the plugin can use the JSON-RPC `listpeers` command to get additional details should they be required. `direction` is either `"in"` or `"out"`. 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.

Note that `peer_connected` is a chained hook. The first plugin that decides to `disconnect` with or without an `error_message` will lead to the subsequent plugins not being called anymore.

5.5.2 `commitment_revocation`

This hook is called whenever a channel state is updated, and the old state was revoked. State updates in Lightning consist of the following steps:

1. Proposal of a new state commitment in the form of a commitment transaction
2. Exchange of signatures for the agreed upon commitment transaction
3. Verification that the signatures match the commitment transaction
4. Exchange of revocation secrets that could be used to penalize an eventual misbehaving party

The `commitment_revocation` hook is used to inform the plugin about the state transition being completed, and deliver the penalty transaction. The penalty transaction could then be sent to a watchtower that automaticaly reacts in case one party attempts to settle using a revoked commitment.

The payload consists of the following information:

```
{
        "commitment_txid":
↪"58eea2cf538cfed79f4d6b809b920b40bb6b35962c4bb4cc81f5550a7728ab05",
        "penalty_tx": "02000000000101...ac00000000",
        "channel_id":
↪"fb16398de93e8690c665873715ef590c038dfac5dd6c49a9d4b61dccfcedc2fb",
        "commitnum": 21
}
```

Notice that the `commitment_txid` could also be extracted from the sole input of the `penalty_tx`, however it is enclosed so plugins don't have to include the logic to parse transactions.

Not included are the `htlc_success` and `htlc_failure` transactions that may also be spending `commitment_tx` outputs. This is because these transactions are much more dynamic and have a predictable timeout, allowing wallets to ensure a quick checkin when the CLTV of the HTLC is about to expire.

The `commitment_revocation` hook is a chained hook, i.e., multiple plugins can register it, and they will be called in the order they were registered in. Plugins should always return `{"result": "continue"}`, otherwise subsequent hook subscribers would not get called.

5.5.3 `db_write`

This hook is called whenever a change is about to be committed to the database, if you are using a SQLITE3 database (the default). This hook will be useless (the `"writes"` field will always be empty) if you are using a PostgreSQL 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!

This hook, unlike all the other hooks, is also strongly synchronous: `lightningd` will stop almost all the other processing until this hook responds.

```
{
  "data_version": 42,
  "writes": [
    "PRAGMA foreign_keys = ON"
  ]
}
```

This hook is intended for creating continuous backups. The intent is that your backup plugin maintains three pieces of information (possibly in separate files): (1) a snapshot of the database, (2) a log of database queries that will bring that snapshot up-to-date, and (3) the previous `data_version`.

`data_version` is an unsigned 32-bit number that will always increment by 1 each time `db_write` is called. Note that this will wrap around on the limit of 32-bit numbers.

`writes` is an array of strings, each string being a database query that modifies the database. If the `data_version` above is validated correctly, then you can simply append this to the log of database queries.

Your plugin **MUST** validate the `data_version`. It **MUST** keep track of the previous `data_version` it got, and:

1. If the new `data_version` is ***exactly*** one higher than the previous, then this is the ideal case and nothing bad happened and we should save this and continue.
2. If the new `data_version` is ***exactly*** the same value as the previous, then the previous set of queries was not committed. Your plugin **MAY** overwrite the previous set of queries with the current set, or it **MAY** overwrite its entire backup with a new snapshot of the database and the current `writes` array (treating this case as if `data_version` were two or more higher than the previous).
3. If the new `data_version` is ***less than*** the previous, your plugin **MUST** halt and catch fire, and have the operator inspect what exactly happend here.
4. Otherwise, some queries were lost and your plugin **SHOULD** recover by creating a new snapshot of the database: copy the database file, back up the given `writes` array, then delete (or atomically `rename` if in

a POSIX filesystem) the previous backups of the database and SQL statements, or you **MAY** fail the hook to abort `lightningd`.

The "rolling up" of the database could be done periodically as well if the log of SQL statements has grown large.

Any response other than `{"result": "continue"}` will cause lightningd to error without committing to the database! This is the expected way to halt and catch fire.

`db_write` is a parallel-chained hook, i.e., multiple plugins can register it, and all of them will be invoked simultaneously without regard for order of registration. The hook is considered handled if all registered plugins return `{"result": "continue"}`. If any plugin returns anything else, `lightningd` will error without committing to the database.

5.5.4 `invoice_payment`

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

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

The hook is deliberately sparse, since the plugin can use the JSON-RPC `listinvoices` command to get additional details about this invoice. It can return a `failure_message` field as defined for final nodes in BOLT 4, a `result` field with the string `reject` to fail it with `incorrect_or_unknown_payment_details`, or a `result` field with the string `continue` to accept the payment.

5.5.5 `openchannel`

This hook is called whenever a remote peer tries to fund a channel to us using the v1 protocol, and it has passed basic sanity checks:

```
{
  "openchannel": {
    "id": "03864ef025fde8fb587d989186ce6a4a186895ee44a926bfc370e2c366597a3f8f",
    "funding_msat": 100000000,
    "push_msat": 0,
    "dust_limit_msat": 546000,
    "max_htlc_value_in_flight_msat": 18446744073709551615,
    "channel_reserve_msat": 1000000,
    "htlc_minimum_msat": 0,
    "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"
        "mindepth": 0,
        "reserve": "1234sat"
}
```

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.

- `mindepth` is the number of confirmations to require before making the channel usable. Notice that setting this to 0 (`zeroconf`) or some other low value might expose you to double-spending issues, so only lower this value from the default if you trust the peer not to double-spend, or you reject incoming payments, including forwards, until the funding is confirmed.
- `reserve` is an absolute value for the amount in the channel that the peer must keep on their side. This ensures that they always have something to lose, so only lower this below the 1% of funding amount if you trust the peer. The protocol requires this to be larger than the dust limit, hence it will be adjusted to be the dust limit if the specified value is below.

Note that `openchannel` is a chained hook. Therefore `close_to`, `reserve` will only be evaluated for the first plugin that sets it. If more than one plugin tries to set a `close_to` address an error will be logged.

5.5.6 `openchannel2`

This hook is called whenever a remote peer tries to fund a channel to us using the v2 protocol, and it has passed basic sanity checks:

```
{
  "openchannel2": {
    "id": "03864ef025fde8fb587d989186ce6a4a186895ee44a926bfc370e2c366597a3f8f",
    "channel_id": "252d1b0a1e57895e84137f28cf19ab2c35847e284c112fefdecc7afeaa5c1de7",
    "their_funding_msat": 100000000,
    "dust_limit_msat": 546000,
    "max_htlc_value_in_flight_msat": 18446744073709551615,
    "htlc_minimum_msat": 0,
    "funding_feerate_per_kw": 7500,
    "commitment_feerate_per_kw": 7500,
    "feerate_our_max": 10000,
    "feerate_our_min": 253,
    "to_self_delay": 5,
    "max_accepted_htlcs": 483,
    "channel_flags": 1
    "locktime": 2453,
    "channel_max_msat": 16777215000,
    "requested_lease_msat": 100000000,
    "lease_blockheight_start": 683990,
    "node_blockheight": 683990
  }
}
```

There may be additional fields, such as `shutdown_scriptpubkey`. You can see the definitions of these fields in BOLT 2's description of the open_channel message.

`requested_lease_msat`, `lease_blockheight_start`, and `node_blockheight` are only present if the opening peer has requested a funding lease, per `option_will_fund`.

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; you can include a `psbt` and an `our_funding_msat` to contribute funds, inputs and outputs to this channel open.

Note that, like `openchannel_init` RPC call, the `our_funding_msat` amount must NOT be accounted for in any supplied output. Change, however, should be included and should use the `funding_feerate_per_kw` to calculate.

See `plugins/funder.c` for an example of how to use this hook to contribute funds to a channel open.

e.g.

```
{
    "result": "continue",
    "close_to": "bc1qlq8srqnz64wgklmqvurv7qnr4rvtq2u96hhfg2"
    "psbt": "cHNidP8BADMCAAAAAQ+yBipSVZrrw28Oed52hTw3N7t0HbIyZhFdcZRH3+61AQAAAAD9////
↪AGYAAAAAAQDfAgAAAAABARtaSZufCbC+P+/G23XVaQ8mDwZQFW1vlCsCYhLbmVrpAAAAAAD+////
↪AvJs5ykBAAAAFgAUT6ORgb3CgFsbwSOzNLzF7jQS5s+AhB4AAAAAABepFNi369DMyAJmqX2agouvGHcDKsZkhwJHMEQCIHELIy
↪PDElnqWw49y2vTqauSJIVBgGtSc+vq5BQd+gEhAg0f8WITWvA8o4grxNKfgdrNDncqreMLeRFiteUlne+GZQAAAAEBIICEHgAAA
↪AlsaWdodG5pbmcCAgABAA==",
    "our_funding_msat": 39999000
}
```

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.

Note that `openchannel` is a chained hook. Therefore `close_to` will only be evaluated for the first plugin that sets it. If more than one plugin tries to set a `close_to` address an error will be logged.

5.5.7 `openchannel2_changed`

This hook is called when we received updates to the funding transaction from the peer.

```
{
        "openchannel2_changed": {
                "channel_id": "252d1b0a1e57895e841...",
                "psbt": "cHNidP8BADMCAAAAAQ+yBipSVZr..."
        }
}
```

In return, we expect a `result` indicated to `continue` and an updated `psbt`. If we have no updates to contribute, return the passed in PSBT. Once no changes to the PSBT are made on either side, the transaction construction negotation will end and commitment transactions will be exchanged.

Expected Return

```
{
        "result": "continue",
        "psbt": "cHNidP8BADMCAAAAAQ+yBipSVZr..."
}
```

See plugins/funder.c for an example of how to use this hook to continue a v2 channel open.

5.5.8 openchannel2_sign

This hook is called after we've gotten the commitment transactions for a channel open. It expects psbt to be returned which contains signatures for our inputs to the funding transaction.

```
{
        "openchannel2_sign": {
                "channel_id": "252d1b0a1e57895e841...",
                "psbt": "cHNidP8BADMCAAAAAQ+yBipSVZr..."
        }
}
```

In return, we expect a result indicated to continue and an partially signed psbt.

If we have no inputs to sign, return the passed in PSBT. Once we have also received the signatures from the peer, the funding transaction will be broadcast.

Expected Return

```
{
        "result": "continue",
        "psbt": "cHNidP8BADMCAAAAAQ+yBipSVZr..."
}
```

See plugins/funder.c for an example of how to use this hook to sign a funding transaction.

5.5.9 rbf_channel

Similar to openchannel2, the rbf_channel hook is called when a peer requests an RBF for a channel funding transaction.

```
{
  "rbf_channel": {
    "id": "03864ef025fde8fb587d989186ce6a4a186895ee44a926bfc370e2c366597a3f8f",
    "channel_id": "252d1b0a1e57895e84137f28cf19ab2c35847e284c112fefdecc7afeaa5c1de7",
    "their_last_funding_msat": 100000000,
    "their_funding_msat": 100000000,
    "our_last_funding_msat": 100000000,
    "funding_feerate_per_kw": 7500,
    "feerate_our_max": 10000,
    "feerate_our_min": 253,
    "channel_max_msat": 16777215000,
    "locktime": 2453,
    "requested_lease_msat": 100000000,
  }
}
```

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 include a psbt and an our_funding_msat to contribute funds, inputs and outputs to this channel open.

Note that, like the `openchannel_init` RPC call, the `our_funding_msat` amount must NOT be accounted for in any supplied output. Change, however, should be included and should use the `funding_feerate_per_kw` to calculate.

Return

```
{
    "result": "continue",
    "psbt": "cHNidP8BADMCAAAAAQ+yBipSVZrrw28Oed52hTw3N7t0HbIyZhFdcZRH3+61AQAAAAD9////
↪AGYAAAAAAQDfAgAAAAABARtaSZufCbC+P+/G23XVaQ8mDwZQFW1vlCsCYhLbmVrpAAAAAAD+////
↪AvJs5ykBAAAAFgAUT6ORgb3CgFsbwSOzNLzF7jQS5s+AhB4AAAAAABepFNi369DMyAJmqX2agouvGHcDKsZkhwJHMEQCIHELIy
↪PDElnqWw49y2vTqauSJIVBgGtSc+vq5BQd+gEhAg0f8WITWvA8o4grxNKfgdrNDncqreMLeRFiteUlne+GZQAAAAEBIICEHgAA
↪AlsaWdodG5pbmcCAgABAA==",
    "our_funding_msat": 39999000
}
```

5.5.10 `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": "",
    "short_channel_id": "1x2x3",
    "forward_msat": 42,
    "outgoing_cltv_value": 500014,
    "shared_secret": "0000000000000000000000000000000000000000000000000000000000000000
↪",
    "next_onion": "[1365bytes of serialized onion]"
  },
  "htlc": {
    "short_channel_id": "4x5x6",
    "id": 27,
    "amount_msat": 43,
    "cltv_expiry": 500028,
    "cltv_expiry_relative": 10,
    "payment_hash": "0000000000000000000000000000000000000000000000000000000000000000"
  },
  "forward_to": "0000000000000000000000000000000000000000000000000000000000000000"
}
```

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.
  - `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:
  - short_channel_id is the channel this payment is coming from.
  - id is the low-level sequential HTLC id integer as sent by the channel peer.
  - 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.
- forward_to: if set, the channel_id we intend to forward this to (will not be present if the short_channel_id was invalid or we were the final destination).

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.

It can also replace the onion.payload by specifying a payload in the response. Note that this is always a TLV-style payload, so unlike onion.payload there is no length prefix (and it must be at least 4 hex digits long). This will be re-parsed; it's useful for removing onion fields which a plugin doesn't want lightningd to consider.

It can also specify forward_to in the response, replacing the destination. This usually only makes sense if it wants to choose an alternate channel to the same next peer, but is useful if the payload is also replaced.

```
{
  "result": "fail",
  "failure_message": "2002"
}
```

fail will tell lightningd to fail the HTLC with a given hex-encoded failure_message (please refer to the spec for details: incorrect_or_unknown_payment_details is the most common).

```
{
  "result": "fail",
  "failure_onion": "[serialized error packet]"
}
```

Instead of `failure_message` the response can contain a hex-encoded `failure_onion` that will be used instead (please refer to the spec for details). This can be used, for example, if you're writing a bridge between two Lightning Networks. Note that `lightningd` will apply the obfuscation step to the value returned here with its own shared secret (and key type `ammag`) before returning it to the previous hop.

```
{
  "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.

The `htlc_accepted` hook is a chained hook, i.e., multiple plugins can register it, and they will be called in the order they were registered in until the first plugin return a result that is not `{"result": "continue"}`, after which the event is considered to be handled. After the event has been handled the remaining plugins will be skipped.

5.5.11 `rpc_command`

The `rpc_command` hook allows a plugin to take over any RPC command. It sends the received JSON-RPC request (for any method!) 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

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

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": {
        }
    }
}
```

Note: The `rpc_command` hook is chainable. If two or more plugins try to replace/result/error the same `method`, only the first plugin in the chain will be respected. Others will be ignored and a warning will be logged.

5.5.12 `custommsg`

The `custommsg` plugin hook is the receiving counterpart to the `sendcustommsg` RPC method and allows plugins to handle messages that are not handled internally. The goal of these two components is to allow the implementation of custom protocols or prototypes on top of a Core Lightning node, without having to change the node's implementation itself.

The payload for a call follows this format:

```
{
        "peer_id": "02df5ffe895c778e10f7742a6c5b8a0cefbe9465df58b92fadeb883752c8107c8f
↪",
        "payload": "1337ffffffff"
}
```

This payload would have been sent by the peer with the `node_id` matching `peer_id`, and the message has type `0x1337` and contents `ffffffff`. Notice that the messages are currently limited to odd-numbered types and must not match a type that is handled internally by Core Lightning. These limitations are in place in order to avoid conflicts with the internal state tracking, and avoiding disconnections or channel closures, since odd-numbered message can be ignored by nodes (see "it's ok to be odd" in the specification for details). The plugin must implement the parsing of the message, including the type prefix, since Core Lightning does not know how to parse the message.

Because this is a chained hook, the daemon expects the result to be `{'result': 'continue'}`. It will fail if something else is returned.

5.5.13 `onion_message_recv` and `onion_message_recv_secret`

(WARNING: experimental-offers only)

These two hooks are almost identical, in that they are called when an onion message is received.

`onion_message_recv` is used for unsolicited messages (where the source knows that it is sending to this node), and `onion_message_recv_secret` is used for messages which use a blinded path we supplied. The latter hook will have a `pathsecret` field, the former never will.

These hooks are separate, because replies MUST be ignored unless they use the correct path (i.e. `onion_message_recv_secret`, with the expected `pathsecret`). This avoids the source trying to probe for responses without using the designated delivery path.

The payload for a call follows this format:

```
{
    "onion_message": {
        "pathsecret":
↪"0000000000000000000000000000000000000000000000000000000000000000",
        "reply_first_node":
↪"02df5ffe895c778e10f7742a6c5b8a0cefbe9465df58b92fadeb883752c8107c8f",
        "reply_blinding":
↪"02df5ffe895c778e10f7742a6c5b8a0cefbe9465df58b92fadeb883752c8107c8f",
                "reply_path": [ {"id":
↪"02df5ffe895c778e10f7742a6c5b8a0cefbe9465df58b92fadeb883752c8107c8f",
                         "encrypted_recipient_data": "0a020d0d",
                         "blinding":
↪"02df5ffe895c778e10f7742a6c5b8a0cefbe9465df58b92fadeb883752c8107c8f"} ],
        "invoice_request": "0a020d0d",
                "invoice": "0a020d0d",
                "invoice_error": "0a020d0d",
                "unknown_fields": [ {"number": 12345, "value": "0a020d0d"} ]
        }
}
```

All fields shown here are optional.

We suggest just returning `{'result': 'continue'}`; any other result will cause the message not to be handed to any other hooks.

5.6 Bitcoin backend

Core Lightning communicates with the Bitcoin network through a plugin. It uses the `bcli` plugin by default but you can use a custom one, multiple custom ones for different operations, or write your own for your favourite Bitcoin data source!

Communication with the plugin is done through 5 JSONRPC commands, `lightningd` can use from 1 to 5 plugin(s) registering these 5 commands for gathering Bitcoin data. Each plugin must follow the below specification for `lightningd` to operate.

5.6.1 `getchaininfo`

Called at startup, it's used to check the network `lightningd` is operating on and to get the sync status of the backend.

The plugin must respond to `getchaininfo` with the following fields: - `chain` (string), the network name as introduced in bip70 - `headercount` (number), the number of fetched block headers - `blockcount` (number), the number of fetched block body - `ibd` (bool), whether the backend is performing initial block download

5.6.2 `estimatefees`

Polled by `lightningd` to get the current feerate, all values must be passed in sat/kVB.

If fee estimation fails, the plugin must set all the fields to `null`.

The plugin, if fee estimation succeeds, must respond with the following fields: - `opening` (number), used for funding and also misc transactions - `mutual_close` (number), used for the mutual close transaction - `unilateral_close` (number), used for unilateral close (/commitment) transactions - `delayed_to_us` (number), used for resolving our output from our unilateral close - `htlc_resolution` (number), used for resolving HTLCs after an unilateral close - `penalty` (number), used for resolving revoked transactions - `min_acceptable` (number), used as the minimum acceptable feerate - `max_acceptable` (number), used as the maximum acceptable feerate

5.6.3 `getrawblockbyheight`

This call takes one parameter, `height`, which determines the block height of the block to fetch.

The plugin must set all fields to `null` if no block was found at the specified `height`.

The plugin must respond to `getrawblockbyheight` with the following fields: - `blockhash` (string), the block hash as a hexadecimal string - `block` (string), the block content as a hexadecimal string

5.6.4 `getutxout`

This call takes two parameter, the `txid` (string) and the `vout` (number) identifying the UTXO we're interested in.

The plugin must set both fields to `null` if the specified TXO was spent.

The plugin must respond to `gettxout` with the following fields: - `amount` (number), the output value in **sats** - `script` (string), the output scriptPubKey

5.6.5 `sendrawtransaction`

This call takes two parameters, a string `tx` representing a hex-encoded Bitcoin transaction, and a boolean `allowhighfees`, which if set means suppress any high-fees check implemented in the backend, since the given transaction may have fees that are very high.

The plugin must broadcast it and respond with the following fields: - `success` (boolean), which is `true` if the broadcast succeeded - `errmsg` (string), if success is `false`, the reason why it failed

CHAPTER 6

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.

6.1 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.

6.2 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.

6.3 Debugging

You can build Core 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.

6.4 Database

Core 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.

6.4.1 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.

6.5 Build and Development

Install the following dependencies for best results:

```
sudo apt update
sudo apt install valgrind cppcheck shellcheck libsecp256k1-dev libpq-dev
```

Re-run `configure` and build using `make`:

```
./configure --enable-developer
make -j$(nproc)
```

6.6 Testing

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 four 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:contrib/pyln-proto 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 Github Actions instance (see `.github/workflows/*.yml`) runs all these for each pull request.

6.6.1 Additional Environment Variables

```
TEST_CHECK_DBSTMTS=[0|1]            - When running blackbox tests, this will
                                      load a plugin that logs all compiled
                                      and expanded database statements.
                                      Note: Only SQLite3.
TEST_DB_PROVIDER=[sqlite3|postgres] - Selects the database to use when running
                                      blackbox tests.
EXPERIMENTAL_DUAL_FUND=[0|1]            - Enable dual-funding tests.
```

6.6.2 Troubleshooting

Valgrind complains about code we don't control

Sometimes `valgrind` will complain about code we do not control ourselves, either because it's in a library we use or it's a false positive. There are generally three ways to address these issues (in descending order of preference):

1. Add a suppression for the one specific call that is causing the issue. Upon finding an issue `valgrind` is instructed in the testing framework to print filters that'd match the issue. These can be added to the suppressions file under `tests/valgrind-suppressions.txt` in order to explicitly skip reporting these in future. This is preferred over the other solutions since it only disables reporting selectively for things that were manually checked. See the valgrind docs for details.
2. Add the process that `valgrind` is complaining about to the `--trace-children-skip` argument in `pyln-testing`. This is used in cases of full binaries not being under our control, such as the `python3` interpreter used in tests that run plugins. Do not use this for binaries that are compiled from our code, as it tends to mask real issues.
3. Mark the test as skipped if running under `valgrind`. It's mostly used to skip tests that otherwise would take considerably too long to test on CI. We discourage this for suppressions, since it is a very blunt tool.

6.7 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`.

6.8 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`.

6.9 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.

6.10 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 first deleting the local copy(if any) at directory .tmp.bolts, setting `BOLTDIR` and `BOLTVERSION` appropriately, and finally running `make extract-bolt-csv`. By default the bolts will be retrieved from the directory `../bolts` and a recent git version.

e.g., `make extract-bolt-csv BOLTDIR=../bolts BOLTVERSION=ee76043271f79f45b3392e629fd35e47f126`

6.11 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 7

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.

7.1 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!

7.2 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.

7.3 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);
}
```

7.4 Tabs and indentaion

The C code uses TAB charaters with a visual indentation of 8 whitespaces. If you submit code for a review, make sure your editor knows this.

When breaking a line with more than 80 characters, align parameters and arguments like so:

```
static void subtract_received_htlcs(const struct channel *channel,
                                    struct amount_msat *amount)
```

Note: For more details, the files `.clang-format` and `.editorconfig` are located in the projects root directory.

7.5 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)
```

7.6 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));
```

7.7 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!

7.8 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!

7.9 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.

7.10 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.

7.11 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).

7.12 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.

7.13 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.

7.14 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.

7.15 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.

7.16 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.

7.17 Creating JSON APIs

Our JSON RPCs always return a top-level object. This allows us to add warnings (e.g. that we're still starting up) or other optional fields later.

Prefer to use JSON names which are already in use, or otherwise names from the BOLT specifications.

The same command should always return the same JSON format: this is why e.g. `listchannels` return an array even if given an argument so there's only zero or one entries.

All `warning` fields should have unique names which start with `warning_`, the value of which should be an explanation. This allows for programs to deal with them sanely, and also perform translations.

7.17.1 Documenting JSON APIs

We use JSON schemas to validate that JSON-RPC returns are in the correct form, and also to generate documentation. See writing schemas manpage.

7.18 Changing JSON APIs

All JSON API changes need a Changelog line (see below).

You can always add a new output JSON field (Changelog-Added), but you cannot remove one without going through a 6-month deprecation cycle (Changelog-Deprecated)

So, only output it if `allow-deprecated-apis` is true, so users can test their code is futureproof. In 6 months remove it (Changelog-Removed).

Changing existing input parameters is harder, and should generally be avoided. Adding input parameters is possible, but should be done cautiously as too many parameters get unwieldy quickly.

7.19 Github Workflows

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

7.19.1 Changelog Entries in Commit Messages

We are maintaining a changelog 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-Experimental:` if it only affects –enable-experimental-features builds, or experimental-config options.

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 8

Release checklist

Here's a checklist for the release process.

8.1 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.

8.2 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`. This does API queries to GitHub, which are severely ratelimited unless you use an API token: set the `GH_TOKEN` environment variable to a Personal Access Token from https://github.com/settings/tokens
3. Create a new CHANGELOG.md heading to `v<VERSION>rc1`, and create a link at the bottom. 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. Update the contrib/pyln package versions: `make update-pyln-versions NEW_VERSION=<VERSION>`

5. Create a PR with the above.

8.3 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 Libera.
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.

8.4 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 Libera.
5. Upgrade your personal nodes to the rc2.

8.5 Tagging the Release

1. Update the CHANGELOG.md; remove -rcN in both places, update the date and add title and namer.
2. Add a PR with that release.
3. Merge the PR, then:
   - `export VERSION=0.9.3`
   - `git pull`
   - `git tag -a -s v${VERSION} -m v${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. 11.`make pyln-release` to upload pyln modules to pypi.org.

8.6 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 Libera.
4. Send a mail to c-lightning and lightning-dev mailing lists, using the same wording as the Release Notes in github.

8.7 Post-release

1. Look through PRs which were delayed for release and merge them.
2. Close out the Milestone for the now-shipped release.
3. Update this file with any missing or changed instructions.

CHAPTER 9

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.

9.1 22.11rc1 - 2022-11-10

9.1.1 Added

- cli: new `--filter` parameter to reduce JSON output. (#5681)
- Documentation: `lightningd-rpc` manual page describes details of our JSON-RPC interface, including compatibility and filtering. (#5681)
- pyln: LightningRpc has new `reply_filter` context manager for reducing output of RPC commands. (#5681)
- JSON-RPC: `filter` object allows reduction of JSON response to (most) commands. (#5681)
- Reckless - a Core Lightning plugin manager (#5647)
- cln-plugin: Options are no longer required to have a default value (#5369)
- Added interactive transaction building routine (#5287)
- cln-rpc: `keysend` now exposes the `extratlvs` field (#5674)
- JSON-RPC: The `extratlvs` argument for `keysend` now allows quoting the type numbers in string (#5674)
- JSON-RPC: `makesecret` can take a string argument instead of hex. (#5633)
- Protocol: We now delay forgetting funding-spent channels for 12 blocks (as per latest BOLTs, to support splicing in future). (#5592)
- Protocol: We now set the `dont_forward` bit on private channel_update's message_flags (as per latest BOLTs). (#5592)

- JSON-RPC: `delpay` takes optional `groupid` and `partid` parameters to specify exactly what payment to delete. (#5594)
- JSON-RPC: `batching` command to allow database transactions to cross multiple back-to-back JSON commands. (#5594)
- Plugins: `autoclean-once` command for a single cleanup. (#5594)
- Plugins: `autoclean-status` command to see what autoclean is doing. (#5594)
- Plugins: `autoclean` can now delete old forwards, payments, and invoices automatically. (#5594)
- JSON-RPC: `delforward` command to delete listforwards entries. (#5594)
- JSON-RPC: `listhtlcs` new command to list all known HTLCS. (#5594)
- JSON-RPC: `listforwards` now shows `in_htlc_id` and `out_htlc_id` (#5594)
- Config: `accept-htlc-tlv-types` lets us accept unknown even HTLC TLV fields we would normally reject on parsing (was EXPERIMENTAL-only `experimental-accept-extra-tlv-types`). (#5619)
- JSON-RPC: `keysend` now has `extratlvs` option in non-EXPERIMENTAL builds. (#5619)
- Protocol: `keysend` will now attach the longest valid text field in the onion to the invoice (so you can have Sphinx.chat users spam you!) (#5619)
- plugin: The `openchannel` hook may return a custom absolute `reserve` value that the peer must not dip below. (#5315)
- JSON-RPC: `fundchannel`, `multifundchannel` and `fundchannel_start` now accept a `reserve` parameter to indicate the absolute reserve to impose on the peer. (#5315)
- Config: `--database-upgrade=true` required if a non-release version wants to (irrevocably!) upgrade the db. (#5550)
- Plugins: Added notification topic "block_processed". (#5581)
- JSON-RPC: `pay` and `listpays` now lists the completion time. (#5398)
- JSON-RPC: `channel_opened` notification `channel_ready` flag. (#5490)

9.1.2 Changed

- JSON-RPC: `listfunds` now lists coinbase outputs as 'immature' until they're spendable (#5664)
- JSON-RPC: UTXOs aren't spendable while immature (#5664)
- Plugins: `openchannel2` now always includes the `channel_max_msat` (#5650)
- JSON-RPC: `createonion` no longer allows non-TLV-style payloads. (#5639)
- cln-plugin: Moved the state binding to the plugin until after the configuration step (#5493)
- Protocol: We now require all channel_update messages include htlc_maximum_msat (as per latest BOLTs) (#5592)
- pyln-spec: package updated to latest spec version. (#5621)
- JSON-RPC: `listforwards` now never shows `payment_hash`; use `listhtlcs`. (#5594)
- cln-rpc: The `wrong_funding` argument for `close` was changed from `bytes` to `outpoint` (#5444)
- JSON-RPC: Error code from bcli plugin changed from 400 to 500. (#5596)
- Plugins: `balance_snapshot` notification does not send balances for channels that aren't locked-in/opened yet (#5587)

- Bolt7 #911 DNS annoucenent support is no longer EXPERIMENTAL (#5487)
- Plugins: RPC operations are now still available during shutdown. (#5577)
- JSON-RPC: `listpeers status` now refers to "channel ready" rather than "funding locked" (BOLT language change for zeroconf channels) (#5490)
- Protocol: `funding_locked` is now called `channel_ready` as per latest BOLTs. (#5490)

9.1.3 Deprecated

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

- JSON-RPC: `autocleaninvoice` (use option `autoclean-expiredinvoices-age`) (#5594)
- JSON-RPC: `delexpiredinvoice`: use `autoclean-once`. (#5594)
- JSON-RPC: `commando-rune` restrictions is always an array, each element an array of alternatives. Replaces a string with `|`-separators, so no escaping necessary except for `\\`. (#5539)
- JSON-RPC: `channel_opened` notification `funding_locked` flag (use `channel_ready`: BOLTs namechange). (#5490)

9.1.4 Removed

- Protocol: we no longer forward HTLCs with legacy onions. (#5639)
- `hsmtool`: hsm_secret (ignored) on cmdline for dumponchaindescriptors (deprecated in v0.9.3) (#5490)
- Plugins: plugin init `use_proxy_always` (deprecated v0.10.2) (#5490)
- JSON-RPC: plugins must supply `usage` parameter (deprecated v0.7) (#5490)
- Old order of the `status` parameter in the `listforwards` rpc command (deprecated in v0.10.2) (#5490)
- JSONRPC: RPC framework now requires the `"jsonrpc"` property inside the request (deprecated in v0.10.2) (#5490)
- JSON API: Removed double wrapping of `rpc_command` payload in `rpc_command` JSON field (deprecated v0.8.2) (#5490)

9.1.5 Fixed

- onchaind: Witness weight estimations could be slightly lower than the VLS signer (#5669)
- Protocol: we now correctly decrypt non-256-length onion errors (we always forwarded them fine, now we actually can parse them). (#5698)
- Fixed gossip_store corruption from duplicate private channel updates (#5661)
- devtools: `mkfunding` command no longer crashes (abort) (#5677)
- Plugins: `funder` now honors lease requests across RBFs (#5650)
- Plugins: `keysend` now removes unknown even (technically illegal!) fields, to try to accept more payments. (#5645)
- channeld: Channel reinitialization no longer fails when the number of outstanding outgoing HTLCs exceeds `max_accepted_htlcs`. (#5640)
- pay: Squeezed out the last `msat` from our local view of the network (#5315)

- Fixed a condition for newly created channels that could trigger a need for reconnect. (#5601)
- peer_control: getinfo shows the correct port on discovered IPs (#5585)
- bcli: don't expose bitcoin RPC password on commandline (#5509)
- Plugins: topology plugin could crash when it sees duplicate private channel announcements. (#5593)
- JSON-RPC: `commando-rune` now handles \ escapes properly. (#5539)
- proper gossip_store operation may resolve some previous gossip propagation issues (#5591)
- peer_control: getinfo showing unannounced addresses. (#5584)

9.1.6 EXPERIMENTAL

- JSON-RPC: `pay` and `sendpay` `localofferid` is now `localinvreqid`. (#5676)
- Protocol: Support for forwarding blinded payments (as per latest draft) (#5646)
- offers: complete rework of spec from other teams (yay!) breaks previous compatibility (boo!) (#5646)
- offers: old `payer_key` proofs won't work. (#5646)
- remove "vendor" (use "issuer") and "timestamp" (use "created_at") fields (deprecated v0.10.2). (#5490)

9.2 0.12.0 - 2022-08-23: Web-8 init

This release named by @adi2011.

Developers please note the Great Msat Migration has begun:

1. All JSON amount field names now end in "_msat" (others are deprecated)
2. Their values are strings ending in "msat", but will soon be normal integers.
3. You should accept both: set `allow-deprecated-apis=false` to test!

9.2.1 Added

- *NEW*: `commando` a new builtin plugin to send/recv peer commands over the lightning network, using runes. (#5370)
- *NEW*: New built-in plugin `bookkeeper` w/ commands `bkpr-listaccountevents`, `bkpr-listbalances`, `bkpr-listincome`, `bkpr-channelsapy`, `bkpr-dumpincomecsv`, `bkpr-inspect` (#5071)
- *NEW*: Emergency channel backup ("static backup") which allows us to seek fund recovery from honest peers in case of complete data loss (#5422)
- Config: `log-level=debug:<partial-nodeid>` supported to get debug-level logs for everything about a peer. (#5349)
- JSON-RPC: `connect` use the standard port derivation when the port is not specified. (#5242)
- JSON-RPC: `fetchinvoice` changes `amount_msat` (#5306)
- JSON-RPC: Added `mindepth` argument to specify the number of confirmations we require for `fundchannel` and `multifundchannel` (#5275)

- JSON-RPC: `listpeers` new fields for `funding` (`remote_funds_msat`, `local_funds_msat`, `fee_paid_msat`, `fee_rcvd_msat`). (#5477)
- JSON-RPC: `listpeers` add optional `remote_addr` (#5244)
- JSON-RPC: `listforwards` now shows `out_channel` in more cases: even if it couldn't actually send to it. (#5330)
- JSON-RPC: `pay` `attempts` `amount_msat` field. (#5306)
- Protocol: private channels will only route using short-channel-ids if channel opened with option_scid_alias-supporting peer. (#5501)
- Protocol: invoice routehints will use fake short-channel-ids for private channels if channel opened with option_scid_alias-supporting peer. (#5501)
- Protocol: we now advertize the `option_channel_type` feature (which we actually supported since v0.10.2) (#5455)
- Plugins: `channel_state_changed` now triggers for a v1 channel's initial "CHANNELD_AWAITING_LOCKIN" state transition (from prior state "unknown") (#5381)
- Plugins: `htlc_accepted_hook` `amount_msat` field. (#5306)
- Plugins: `htlc_accepted` now exposes the `short_channel_id` for the channel from which that HTLC is coming from and the low-level per-channel HTLC `id`, which are necessary for bridging two different Lightning Networks when MPP is involved. (#5303)
- Plugins: The `openchannel` hook may return a `mindepth` indicating how many confirmations are required. (#5275)
- msggen: introduce chain of responsibility pattern to make msggen extensible (#5216)
- cln_plugin: persist cln configuration from init msg (#5279)
- pyln-testing: Added utilities to read and parse `gossip_store` file for nodes. (#5275)
- `hsmtool`: new command `checkhsm` to check BIP39 passphrase against hsm_secret. (#5441)
- contrib: Added `fund_ln` to the contrib/startup_regtest.sh (#5062)
- build: Added m1 architecture support for macos (#4988)
- build: Reproducible builds now include rust binaries such as the `cln-grpc` plugin (#5421)

9.2.2 Changed

- `lightningd`: will refuse to start with the wrong node_id (i.e. hsm_secret changes). (#5425)
- `connectd`: prefer IPv6 connections when available. (#5244)
- `connectd`: Only use IP discovery as fallback when no addresses would be announced (#5344)
- `connectd`: give busy peers more time to respond to pings. (#5347)
- `gossipd`: now accepts spam gossip, but squelches it for (#5239)
- gossip: gossip_store updated to version 10. (#5239)
- Options: `log-file` option specified multiple times opens multiple log files. (#5281)
- JSON-RPC: `sendpay` and `sendonion` now obey the first hop "channel" short_channel_id, if specified. (#5505)
- JSON-RPC: `signpsbt` no longer crashes if it doesn't like what your PSBT is (#5506)

- JSON-RPC: `signpsbt` will now add redeemscript + witness-utxo to the PSBT for an input that we can sign for, before signing it. (#5506)
- JSON-RPC: `plugin start` now assumes relative path to default plugins dir if the path is not found in absolute context. i.e. lightning-cli plugin start my_plugin.py (#5211)
- JSON-RPC: `fundchannel`: now errors if you try to buy a liquidity ad but dont' have `experimental-dual-fund` enabled (#5389)
- JSON-RPC: "_msat" fields can be raw numbers, not "123msat" strings (please handle both!) (#5306)
- JSON-RPC: `invoice`, `sendonion`, `sendpay`, `pay`, `keysend`, `fetchinvoice`, `sendinvoice`: `msatoshi` argument is now called `amount_msat` to match other fields. (#5306)

9.2.3 Deprecated

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

- JSON-RPC: `listpeers.funded` fields `local_msat` and `remote_msat`. (#5477)
- JSON-RPC: `listtransactions` `msat` (use `amount_msat`) (#5306)
- JSON-RPC: checkmessage return an error when the pubkey is not specified and it is unknown in the network graph. (#5252)
- JSON-RPC: "_msat" fields as "123msat" strings (will be only numbers) (#5306)
- JSON-RPC: `sendpay` `route` elements `msatoshi` (use `amount_msat`) (#5306)
- JSON-RPC: `pay`, `decode`, `decodepay`, `getroute`, `listinvoices`, `listpays` and `listsendpays` `msatoshi` fields (use `amount_msat`). (#5306)
- JSON-RPC: `getinfo` `msatoshi_fees_collected` field (use `fees_collected_msat`). (#5306)
- JSON-RPC: `listpeers` `channels`: `msatoshi_to_us`, `msatoshi_to_us_min`, `msatoshi_to_us_max`, `msatoshi_total`, `dust_limit_satoshis`, `our_channel_reserve_satoshis`, `their_channel_reserve_satoshis`, `spendable_msatoshi`, `receivable_msatoshi`, `in_msatoshi_offered`, `in_msatoshi_fulfilled`, `out_msatoshi_offered`, `out_msatoshi_fulfilled`, `max_htlc_value_in_flight_msat` and `htlc_minimum_msat` (use `to_us_msat`, `min_to_us_msat`, `max_to_us_msat`, `total_msat`, `dust_limit_msat`, `our_reserve_msat`, `their_reserve_msat`, `spendable_msat`, `receivable_msat`, `in_offered_msat`, `in_fulfilled_msat`, `out_offered_msat`, `out_fulfilled_msat`, `max_total_htlc_in_msat` and `minimum_htlc_in_msat`). (#5306)
- JSON-RPC: `listinvoices` and `pay` `msatoshi_received` and `msatoshi_sent` (use `amount_received_msat`, `amount_sent_msat`) (#5306)
- JSON-RPC: `listpays` and `listsendpays` `msatoshi_sent` (use `amount_sent_msat`) (#5306)
- JSON-RPC: `listforwards` `in_msatoshi`, `out_msatoshi` and `fee` (use `in_msat`, `out_msat` and `fee_msat`) (#5306)
- JSON-RPC: `listfunds` `outputs` `value` (use `amount_msat`) (#5306)
- JSON-RPC: `fetchinvoice` `changes` `msat` (use `amount_msat`) (#5306)
- JSON-RPC: `pay` `attempts` `amount` field (use `amount_msat`). (#5306)
- JSON-RPC: `invoice`, `sendonion`, `sendpay`, `pay`, `keysend`, `fetchinvoice`, `sendinvoice` `msatoshi` (use `amount_msat`) (#5306)
- `listconfigs` `plugins` `options` which are not set are omitted, not `null`. (#5306)

- Plugins: `htlc_accepted_hook` `amount` field (use `amount_msat`) (#5306)
- Plugins: `coin_movement` notification: `balance`, `credit`, `debit` and `fees` (use `balance_msat`, `credit_msat`, `debit_msat` and `fees_msat`) (#5306)
- Plugins: `rbf_channel` and `openchannel2` hooks `their_funding` (use `their_funding_msat`) (#5306)
- Plugins: `openchannel2` hook `dust_limit_satoshis` (use `dust_limit_msat`) (#5306)
- Plugins: `openchannel` hook `funding_satoshis` (use `funding_msat`) (#5306)
- Plugins: `openchannel` hook `dust_limit_satoshis` (use `dust_limit_msat`) (#5306)
- Plugins: `openchannel` hook `channel_reserve_satoshis` (use `channel_reserve_msat`) (#5306)
- Plugins: `channel_opened` notification `amount` (use `funding_msat`) (#5306)
- Plugins: `htlc_accepted` `forward_amount` (use `forward_msat`) (#5306)

9.2.4 Removed

- Protocol: We no longer create gossip messages which use zlib encoding (we still understand them, for now!) (#5226)
- JSON-RPC: `getsharedsecret` API: use `makesecret` (#5430)
- JSON-RPC: removed `listtransactions` `outputs` `satoshis` field (deprecated v0.10.1) (#5264)
- JSON-RPC: removed `listpeers` `channels` deprecated fields (deprecated v0.10.1) (#5264)
- JSON-RPC: removed `listpeers` `channels` `closer` now omitted, rather than `null` (deprecated v0.10.1) (#5264)
- libhsmd: Removed the `libhsmd_python` wrapper as it was unused (#5415)
- Options: removed `enable-autotor-v2-mode` option (deprecated v0.10.1) (#5264)

9.2.5 Fixed

- db: postgresql crash on startup when dual-funding lease open is pending with "s32 field doesn't match size: expected 4, actual 8" (#5513)
- `connectd`: various crashes and issues fixed by simplification and rewrite. (#5261)
- `connectd`: Port of a DNS announcement can be 0 if unspecified (#5434)
- `dualopend`: Issue if the number of outputs decreases in a dualopen RBF or splice. (#5378)
- `channeld`: Enforce our own `minimum_depth` beyond just confirming (#5275)
- logging: `log-prefix` now correctly prefixes *all* log messages. (#5349)
- logging: `log-level io` shows JSONRPC output, as well as input. (#5306)
- PSBT: Fix signature encoding to comply with BIP-0171. (#5307)
- signmessage: improve the UX of the rpc command when zbase is not a valid one (#5297)
- JSON-RPC: Adds dynamically detected public IP addresses to `getinfo` (#5244)
- cln-rpc: naming mismatch for `ConnectPeer` causing `connectpeer` to be called on the JSON-RPC (#5362)
- pyln-spec: update the bolts implementation (#5168)

- Plugins: setting the default value of a parameter to `null` is the same as not setting it (pyln plugins did this!). (#5460)
- Plugins: plugins would hang indefinitely despite `lightningd` closing the connection (#5362)
- Plugins: `channel_opened` notification `funding_locked` field is now accurate: was always `true`. (#5489)
- Upgrade docker base image from Debian buster to bullseye to work with glibc 2.29+ #5276 (#5278)
- docker: The docker images are now built with the rust plugins `cln-grpc` (#5270)

9.3 0.11.2 - 2022-06-24: Simon's Carefully Chosen Release Name III

Regressions since 0.10.2 which could not wait for the 0.12 release, which especially hurt larger nodes.

9.3.1 Fixed

- Protocol: treat LND "internal error" as warnings, not force close events (like v0.10) (#5326)
- connectd: no longer occasional crashes when peers reconnect. (#5300)
- connectd: another crash fix on trying to reconnect to disconnecting peer. (#5340)
- topology: Under some circumstances we were considering the limits on the wrong direction for a channel (#5286)
- routing: Fixed an issue where we would exclude the entire channel if either direction was disabled, or we hadn't seen an update yet. (#5286)
- connectd: large memory usage with many peers fixed. (#5312)
- connectd: reduce initial CPU load when connecting to peers. (#5328)
- lightnind: fix failed startup "Could not load channels from the database" if old TORv2 addresses were present. (#5331)

9.4 0.11.1 - 2022-05-13: Simon's Carefully Chosen Release Name II

Single change which fixed a bug introduced in 0.11.0 which could cause unwanted unilateral closes (`bad reestablish revocation_number: 0 vs 3`)

9.4.1 Fixed

- connectd: make sure we don't keep stale reconnections around. (#5256)
- connectd: fix assert which we could trigger. (#5256)

9.5 0.11.0.1 - 2022-04-04: Simon's Carefully Chosen Release Name

This release would have been named by Simon Vrouwe, had he responded to my emails!

This marks the name change to core-lightning (#CLN).

9.5.1 Added

- Protocol: we now support opening multiple channels with the same peer. (#5078)
- Protocol: we send/receive IP addresses in `init`, and send updated node_announcement when two peers report the same remote_addr (`disable-ip-discovery` suppresses this announcement). (#5052)
- Protocol: we more aggressively send our own gossip, to improve propagation chances. (#5200)
- Plugins: `cln-grpc` first class GRPC interface for remotely controlling nodes over mTLS authentication; set `grpc-port` to activate (#5013)
- Database: With the `sqlite3://` scheme for `--wallet` option, you can now specify a second file path for real-time database backup by separating it from the main file path with a `:` character. (#4890)
- Protocol: `pay` (and decode, etc) supports bolt11 payment_metadata a-la https://github.com/lightning/bolts/pull/912 (#5086)
- JSON-RPC: `invoice` has a new parameter `deschashonly` to put hash of description in bolt11. (#5121)
- JSON-RPC: `pay` has new parameter `description`, will be required if bolt11 only has a hash. (#5122)
- JSON-RPC: `pay` has new parameter `maxfee` for setting absolute fee (instead of using `maxfeepercent` and/or `exemptfee`) (#5122)
- JSON-RPC: `listforwards` has new entry `style`, currently "legacy" or "tlv". (#5146)
- JSON-RPC: `delinvoice` has a new parameter `desconly` to remove description. (#5121)
- JSON-RPC: new `setchannel` command generalizes `setchannelfee`: you can now alter the `htlc_minimum_msat` and `htlc_maximum_msat` your node advertizes. (#5103)
- Config: `htlc-minimum-msat` and `htlc-maximum-msat` to set default values to advertizes for new channels. (#5136)
- JSON-RPC: `listpeers` now includes a `pushed_msat` value. For leased channels, is the total lease_fee. (#5043)
- JSON-RPC: `getinfo` result now includes `our_features` (bits) for various Bolt #9 contexts (#5047)
- Docker build for ARM defaults to `bitcoin`, but can be overridden with the `LIGHTNINGD_NETWORK` envvar. (#4896)
- Developer: A new Rust library called `cln-rpc` can be used to interact with the JSON-RPC (#5010)
- JSON-RPC: A new `msggen` library allows easy generation of language bindings for the JSON-RPC from the JSON schemas (#5010)
- JSON-RPC: `listchannels` now includes the `funding_outnum` (#5016)
- JSON-RPC: `coin_movement` to 'external' accounts now include an 'originating_account' field (#5019)
- JSON-RPC: Add `exclude` option for `pay` command to manually exclude channels or nodes when finding a route. (#4906)
- Database: Speed up loading of pending HTLCs during startup by using a partial index. (#4925)

9.5.2 Changed

- JSON-RPC: `close` by peer id will fail if there is more than one live channel (use `channel_id` or `short_channel_id` as id arg). (#5078)
- JSON_RPC: `sendcustommsg` now works with any connected peer, even when shutting down a channel. (#4985)

- JSON_RPC: `ping` now works with connected peers, even without a channel. (#4985)
- cli: Addition of HSM specific error code in lightning-cli (#4908)
- config: If the port is unspecified, the default port is chosen according to used network similarly to Bitcoin Core. (#4900)
- Plugins: `shutdown` notification is now send when lightningd is almost completely shutdown, RPC calls then fail with error code -5. (#4897)
- Protocol: `signet` addresses and invoices now use `tbs` instead of `tb`. (#4929)

9.5.3 Deprecated

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

- JSON-RPC: `pay` for a bolt11 which uses a `description_hash`, without setting `description`. (#5122)
- JSON-RPC: `invoice` `expiry` no longer allowed to be a string with suffix, use an integer number of seconds. (#5104)
- JSON-RPC: `fundpsbt`/`utxopsbt` `reserve` must be a number, not bool (for `true` use 72/don't specify, for `false` use 0). Numbers have been allowed since v0.10.1. (#5104)
- JSON-RPC: `shutdown` no longer allows p2pkh or p2sh addresses. (#5086)
- JSON-RPC: `sendpay` `route` argument `style` "legacy" (don't use it at all, we ignore it now and always use "tlv" anyway). (#5120)
- JSON-RPC: `setchannelfee` (use `setchannel`). (#5103)

9.5.4 Removed

- JSON-RPC: `legacypay` (`pay` replaced it in 0.9.0). (#5122)
- Protocol: support for legacy onion format removed, since everyone supports the new one. (#5058)
- Protocol: ... but we still forward legacy HTLC onions for now. (#5146)
- Plugins: The `message` field on the `custommsg` hook (deprecated in v0.10.0) (#4902)
- JSON-RPC: `fundchannel_complete` `txid` and `txout` parameters (deprecated in v0.10.0) (#4902)

9.5.5 Fixed

- onchaind: we sometimes failed to close upstream htlcs if more than one HTLC is in flight during unilateral close. (#5130)
- JSON-RPC: `listpays` always includes `bolt11` or `bolt12` field. (#5122)
- cli: don't ask to confirm the password if the `hsm_secret` is already encrypted. (#5085)
- cli: check if the `hsm_secret` password and the confirmation match from the command line (#5085)
- JSON-RPC: `connect` notification now called even if we already have a live channel. (#5078)
- docker: The docker image is now built with postgresql support (#5081)
- hsmd: Fixed a significant memory leak (#5051)
- closingd: more accurate weight estimation helps mutual closing near min/max feerates. (#5004)

- Protocol: Always flush sockets to increase chance that final message get to peer (esp. error packets). (#4984)
- JSON-RPC: listincoming showed incoming_capacity_msat field 1000 times actual value. (#4913)
- Options: Respect –always-use-proxy AND –disable-dns when parsing wireaddresses to listen on. (#4829)
- lightningd: remove slow memory leak in DEVELOPER builds. (#4931)
- JSON-RPC: `paystatus` entries no longer have two identical `amount_msat` entries. (#4911)
- We really do allow providing multiple addresses of the same type. (#4902)

9.5.6 EXPERIMENTAL

- Fixed `experimental-websocket` intermittent read errors (#5090)
- Fixed `experimental-websocket-port` not to leave zombie processes. (#5101)
- Config option `--lease-fee-base-msat` renamed to `--lease-fee-base-sat` (#5047)
- Config option `--lease-fee-base-msat` deprecated and will be removed next release (#5047)
- Fixed `experimental-websocket-port` to work with default addresses. (#4945)
- Protocol: removed support for v0.10.1 onion messages. (#4921)
- Protocol: Ability to announce DNS addresses (#4829)
- Protocol: disabled websocket announcement due to LND propagation issues (#5200)

9.6 0.10.2 - 2021-11-03: Bitcoin Dust Consensus Rule

This release named by @vincenzopalazzo.

9.6.1 Added

- config: new option `--max-dust-htlc-exposure-msat`, which limits the total amount of sats to be allowed as dust on a channel (#4837)
- With `sqlite3` db backend we now use a 60-second busy timer, to allow backup processes like `litestream` to operate safely. (#4867)
- pay: Payment attempts are now grouped by the pay command that initiated them (#4567)
- JSON-RPC: `setchannelfee` gives a grace period (`enforcedelay`) before rejecting old-fee payments: default 10 minutes. (#4806)
- Support filtering `listpays` by their status. (#4595)
- `close` now notifies about the feeranges each side uses. (#4784)
- Protocol: We now send and support `channel_type` in channel open (not dual-funding though). (#4616)
- Protocol: We now perform quick-close if the peer supports it. (#4599)
- JSONRPC: `close` now takes a `feerange` parameter to set min/max fee rates for mutual close. (#4599)
- Protocol: Allow sending large HTLCs if peer offers `option_support_large_channel` (> 4294967295msat) (#4599)
- pyln-client: routines for direct access to the gossip store as Gossmap (#4582)

- Plugins: `shutdown` notification for clean exits. (#4754)
- Plugins: Added `channel_id` and `commitnum` to `commitment_revocation` hook (#4760)
- JSON-RPC: `datastore`, `deldatastore` and `listdatastore` for plugins to store simple persistent key/value data. (#4674)

9.6.2 Changed

- pay: The route selection will now use the log-propability-based channel selection to increase success rate and reduce time to completion (#4771)
- Plugins: `pay` now biases towards larger channels, improving success probability. (#4771)
- db: removal of old HTLC information and vacuuming shrinks large lightningd.sqlite3 by a factor of 2-3. (#4850)
- JSON-RPC: `ping` now only works if we have a channel with the peer. (#4804)
- Protocol: Send regular pings to detect dead connections (particularly for Tor). (#4804)
- Build: Python is now required to build, as generated files are no longer checked into the repository. (#4805)
- pyln-spec: updated to latest BOLT versions. (#4763)
- JSON-RPC: Change order parameters in the `listforwards` command (#4668)
- db: We now set a busy timeout to safely allow others to access sqlite3 db (e.g. litestream) (#4554)
- connectd: Try non-TOR connections first (#4731)

9.6.3 Deprecated

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

- Protocol: No longer restrict HTLCs to less than 4294967295msat (#4599)
- Change order of the `status` parameter in the `listforwards` rpc command. (#4668)
- RPC framework now requires the `"jsonrpc"` property inside the request. (#4742)
- Plugins: Renames plugin init `use_proxy_always` to `always_use_proxy` (#4731)

9.6.4 Removed

9.6.5 Fixed

- peer: Fixed a crash when a connection is lost outside of a DB transaction (#4894)
- We now no longer self-limit the number of file descriptors (which limits the number of channels) in sufficiently modern systems, or where we can access `/proc` or `/dev/fd`. We still self-limit on old systems where we cannot find the list of open files on `/proc` or `/dev/fd`, so if you need > ~4000 channels, upgrade or mount `/proc`. (#4872)
- errors: Errors returning a `channel_update` no longer return an outdated one. (#4876)
- pay: `listpays` returns payments orderd by their creation date (#4567)
- pay: `listpays` no longer groups attempts from multiple attempts to pay an invoice (#4567)
- sqlite3: Relaxed the version match requirements to be at least a minimum version and a major version match (#4852)

- pay: `pay` would sometimes misreport a final state of `pending` instead of `failed` (#4803)
- Plugins: C plugins would could leak memory on every command (esp. seen when hammering topology's `listchannels`). (#4737)
- libplugin: Fatal error messages from `plugin_exit()` now logged in lightningd. (#4754)
- `openchannel_signed` would fail on PSBT comparison of materially identical PSBTs (#4752)
- doc: `listnodes` fields now correctly documented. (#4750)
- EXPERIMENTAL: crash for some users while requesting dual funding leases. (#4751)
- Plugins: Don't drop complaints about silly channels to `stderr`. (#4730)
- connectd: Do not try address hint twice (#4731)

9.6.6 EXPERIMENTAL

- channel_upgrade draft upgraded: cannot upgrade channels until peers also upgrade. (#4830)
- bolt12: `chains` in `invoice_request` and invoice is deprecated, `chain` is used instead. (#4849)
- bolt12: `vendor` is deprecated: the field is now called `issuer`. (#4849)
- Protocol: Updated `onion_message` support to match updated draft specification (with backwards compat for old version) (#4800)
- Anchor output mutual close allow a fee higher than the final commitment transaction (as per lightning-rfc #847) (#4599)

9.7 0.10.1 - 2021-08-09: "eltoo: Ethereum Layer Too"

This release named by @nalinbhardwaj.

NOTE ONE: Both the dual-funding and offers protocols have changed, and are incompatible with older releases (they're both still draft) #reckless

NOTE TWO: `rebalance` and `drain` plugins will need to be redownloaded as older versions will no longer work – `payment_secret` is now compulsory.

9.7.1 Added

- JSON-RPC: `invoice` now outputs explicit `payment_secret` as its own field. (#4646)
- JSON-RPC: `listchannels` can be queried by `destination`. (#4614)
- JSON-RPC: `invoice` now gives `warning_private_unused` if unused unannounced channels could have provided sufficient capacity. (#4585)
- JSON-RPC: `withdraw`, `close` (and others) now accept taproot (and other future) segwit addresses. (#4591)
- JSON-RPC: HTLCs in `listpeers` are now annotated with a status if they are waiting on an `htlc_accepted` hook of a plugin. (#4580)
- JSON-RPC: `close` returns `type` "unopened" if it simply discards channel instead of empty object. (#4501)
- JSON-RPC: `listfunds` has a new `reserved_to_block` field. (#4510)
- JSON-RPC: `createonion` RPC command now accepts an optional `onion_size`. (#4519)

- JSON-RPC: new command `parsefeerate` which takes a feerate string and returns the calculated perkw/perkb (#4639)
- Protocol: `option_shutdown_anysegwit` allows future segwit versions on shutdown transactions. (#4556)
- Protocol: We now send and accept `option_shutdown_anysegwit` so you can close channels to v1+ segwit addresses. (#4591)
- Plugins: Plugins may now send custom notifications that other plugins can subscribe to. (#4496)
- Plugins: Add `funder` plugin, which allows you to setup a policy for funding v2 channel open requests. Requres –experimental-dual-fund option (#4489)
- Plugins: `funder` plugin includes command `funderupdate` which will show current funding configuration and allow you to modify them (#4489)
- Plugins: Restart plugin on `rescan` when binary was changed. (#4609)
- keysend: `keysend` can now reach non-public nodes by providing the `routehints` argument if they are known. (#4611)
- keysend: You can now add extra TLVs to a payment sent via `keysend` (#4610)
- config: `force_feerates` option to allow overriding feerate estimates (mainly for regtest). (#4629)
- config: New option `log-timestamps` allow disabling of timestamp prefix in logs. (#4504)
- hsmtool: allow piped passwords (#4571)
- libhsmd: Added python bindings for `libhsmd` (#4498)
- libhsmd: Extracted the `hsmd` logic into its own library for other projects to use (#4497)
- lightningd: we now try to restart if subdaemons are upgraded underneath us. (#4471)

9.7.2 Changed

- JSON-RPC: `invoice` now allows creation of giant invoices (>= 2^32 msat) (#4606)
- JSON-RPC: `invoice` warnings are now better defined, and `warning_mpp_capacity` is no longer included (since `warning_capacity` covers that). (#4585)
- JSON-RPC: `getroute` is now implemented in a plugin. (#4585)
- JSON-RPC: `sendonion` no longer requires the gratuitous `direction` and `channel` fields in the `firsthop` parameter. (#4537)
- JSON-RPC: moved dev-sendcustommsg to sendcustommsg (#4650)
- JSON-RPC: `listpays` output is now ordered by the `created_at` timestamp. (#4518)
- JSON-RPC: `listsendpays` output is now ordered by `id`. (#4518)
- JSON-RPC: `autocleaninvoice` now returns an object, not a raw string. (#4501)
- JSON-RPC: `fundpsbt` will not include UTXOs that aren't economic (can't pay for their own fees), unless 'all' (#4509)
- JSON-RPC: `close` now always returns notifications on delays. (#4465)
- Protocol: All new invoices require a `payment_secret` (i.e. modern TLV format onion) (#4646)
- Protocol: Allow out-of-bound fee updates from peers, as long as they're not getting *worse* (#4681)
- Protocol: We can no longer connect to peers which don't support `payment_secret`. (#4646)

- Protocol: We will now reestablish and negotiate mutual close on channels we've already closed (great if peer has lost their database). (#4559)
- Protocol: We now assume nodes support TLV onions (non-legacy) unless we have a `node_announcement` which says they don't. (#4646)
- Protocol: Use a more accurate fee for mutual close negotiation. (#4619)
- Protocol: channel feerates reduced to bitcoind's "6 block ECONOMICAL" rate. (#4507)
- keysend now uses 22 for the final CLTV, making it rust-lightning compatible. (#4548)
- Plugins: `fundchannel` and `multifundchannel` will now reserve funding they use for 2 weeks instead of 12 hours. (#4510)
- Plugins: we now always send `allow-deprecated-apis` in getmanifest. (#4465)

9.7.3 Deprecated

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

- lightningd: `enable-autotor-v2-mode` option. Use v3. See https://blog.torproject.org/v2-deprecation-timeline. (#4549)
- lightningd: v2 Tor addresses. Use v3. See https://blog.torproject.org/v2-deprecation-timeline. (#4549)
- JSON-RPC: `listtransactions` `outputs` `satoshis` field (use `msat` instead). (#4594)
- JSON-RPC: `listfunds` `channels` `funding_allocation_msat` and `funding_msat`: use `funding`. (#4594)
- JSON-RPC: `listfunds` `channels` `last_tx_fee`: use `last_tx_fee_msat`. (#4594)
- JSON-RPC: `listfunds` `channels` `closer` is now omitted if it does not apply, not JSON `null`. (#4594)

9.7.4 Removed

- JSON-RPC: `newaddr` no longer includes `address` field (deprecated in 0.7.1) (#4465)
- pyln: removed deprecated `fundchannel`/`fundchannel_start` `satoshi` arg. (#4465)
- pyln: removed deprecated pay/sendpay `description` arg. (#4465)
- pyln: removed deprecated close `force` variant. (#4465)

9.7.5 Fixed

- JSON-RPC: `listinvoice` no longer crashes if given an invalid (or bolt12) `invstring` argument. (#4625)
- JSON-RPC: `listconfigs` would list some boolean options as strings `"true"` or `"false"` instead of using JSON booleans. (#4594)
- Protocol: don't ever send 0 `fee_updates` (regtest bug). (#4629)
- Protocol: We could get stuck on signature exchange if we needed to retransmit the final `revoke_and_ack`. (#4559)
- Protocol: Validate chain hash for `gossip_timestamp_filter` messages (#4514)
- Protocol: We would sometimes gratuitously disconnect 30 seconds after an HTLC failed. (#4550)
- Protocol: handle complex feerate transitions correctly. (#4480)

- Protocol: Don't create more than one feerate change at a time, as this seems to desync with LND. (#4480)
- Build: Fixes `make full-check` errors on macOS (#4613)
- Build: Fixes `make` with `--enable-developer` option on macOS. (#4613)
- Docs: Epic documentation rewrite: each now lists complete and accurate JSON output, tested against testsuite. (#4594)
- Config: `addr` autotor and statictor /torport arguments now advertized correctly. (#4603)
- pay: Fixed an issue when filtering routehints when we can't find ourselves in the local network view. (#4581)
- pay: The presplitter mod will no longer exhaust the HTLC budget. (#4563)
- pay: Fix occasional crash paying an invoice with a routehint to us. (#4555)
- Compat: Handle windows-style newlines and other trailing whitespaces correctly in bitcoin-cli interface (#4502)

9.7.6 EXPERIMENTAL

- bolt12 decode `timestamp` field deprecated in favor of new name `created_at`. (#4669)
- JSON-RPC: `listpeers` now includes the `scratch_txid` for every inflight (if is a dual-funded channel) (#4521)
- JSON-RPC: for v2 channels, we now list the inflights information for a channel (#4521)
- JSON-RPC: `fetchinvoice` can take a payer note, and `listinvoice` will show the `payer_notes` received. (#4625)
- JSON-RPC: `fetchinvoice` and `sendinvoice` will connect directly if they can't find an onionmessage route. (#4625)
- JSON-RPC: `openchannel_init` now takes a `requested_amt`, which is an amount to request from peer (#4639)
- JSON-RPC: `fundchannel` now takes optional `request_amt` parameter (#4639)
- JSON-RPC: `fundchannel`, `multifundchannel`, and `openchannel_init` now accept a `compact_lease` for any requested funds (#4639)
- JSON-RPC: close now has parameter to force close a leased channel (`option_will_fund`) (#4639)
- JSON-RPC: `listnodes` now includes the `lease_rates`, if available (#4639)
- JSON-RPC: new RPC `setleaserates`, for passing in the rates to advertise for a channel lease (`option_will_fund`) (#4639)
- JSON-RPC: `decode` now gives a `valid` boolean (it does partial decodes of some invalid data). (#4501)
- JSON-RPC: `listoffers` now shows `local_offer_id` when listing all offers. (#4625)
- Protocol: we can now upgrade old channels to `option_static_remotekey`. See https://github.com/lightningnetwork/lightning-rfc/pull/868 (#4532)
- Protocol: we support the quiescence protocol from https://github.com/lightningnetwork/lightning-rfc/pull/869 (#4520)
- Protocol: Replaces `init_rbf`'s `fee_step` for RBF of v2 opens with `funding_feerate_perkw`, breaking change (#4648)
- Protocol: BOLT12 offers can now be unsigned, for really short QR codes. (#4625)
- Protocol: offer signature format changed. (#4630)

- Plugins: `rbf_channel` hook has `channel_max_msat` parameter (#4489)
- Plugins: `openchannel2` hook now includes optional fields for a channel lease request (#4639)
- Plugins: add a `channel_max_msat` value to the `openchannel2` hook. Tells you the total max funding this channel is allowed to have. (#4489)
- funder: `funderupdate` command to view and update params for contributing our wallet funds to v2 channel openings. Provides params for enabling `option_will_fund`. (#4664)

9.8 0.10.0 - 2021-03-28: Neutralizing Fee Therapy

This release named by @jsarenik.

9.8.1 Added

- Protocol: we treat error messages from peer which refer to "all channels" as warnings, not errors. (#4364)
- Protocol: we now report the new (draft) warning message. (#4364)
- JSON-RPC: `connect` returns `address` it actually connected to (#4436)
- JSON-RPC: `connect` returns "direction" ("in": they initiated, or "out": we initiated) ([#4452])
- JSON-RPC: `txprepare` and `withdraw` now return a `psbt` field. (#4428)
- JSON-RPC: `fundchannel_complete` takes a psbt parameter. (#4428)
- pay: `pay` will now remove routehints that are unusable due to the entrypoint being unknown or unreachable. (#4404)
- Plugins: `peer_connected` hook and `connect` notifications have "direction" field. ([#4452])
- Plugins: If there is a misconfiguration with important plugins we now abort early with a more descriptive error message. (#4418)
- pyln: Plugins that are run from the command line print helpful information on how to configure c-lightning to include them and print metadata about what RPC methods and options are exposed. (#4419)
- JSON-RPC: `listpeers` now shows latest feerate and unilateral close fee. (#4407)
- JSON-RPC: `listforwards` can now filter by status, in and out channel. (#4349)
- JSON-RPC: Add new parameter `excess_as_change` to fundpsbt+utxopsbt (#4368)
- JSON-RPC: `addgossip` allows direct injection of network gossip messages. (#4361)
- pyln-testing: The RPC client will now pretty-print requests and responses to facilitate log-based debugging. (#4357)

9.8.2 Changed

- Plugins: the `rpc_command` hook is now chainable. (#4384)
- JSON-RPC: If bitcoind won't give a fee estimate in regtest, use minimum. (#4405)
- Protocol: we use `sync_complete` for gossip range query replies, with detection for older spec nodes. (#4389)
- Plugins: `peer_connected` rejections now send a warning, not an error, to the peer. (#4364)
- Protocol: we now send warning messages and close the connection, except on unrecoverable errors. (#4364)

- JSON-RPC: `sendpay` no longer extracts updates from errors, the caller should do it from the `raw_message`. (#4361)
- Plugins: `peer_connected` hook is now chainable (#4351)
- Plugins: `custommsg` hook is now chainable (#4358)

9.8.3 Deprecated

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

- JSON-RPC: `fundchannel_complete` `txid` and `txout` parameters (use `psbt`) (#4428)
- Plugins: The `message` field on the `custommsg` hook is deprecated in favor of the `payload` field, which skips the internal prefix. (#4394)

9.8.4 Removed

- `bcli` replacements must allow `allowhighfees` argument (deprecated 0.9.1). (#4362)
- `listsendpays` will no longer add `amount_msat` `null` (deprecated 0.9.1). (#4362)

9.8.5 Fixed

- Protocol: overzealous close when peer sent more HTLCs than they'd told us we could send. (#4432)
- pay: Report the correct decoding error if bolt11 parsing fails. (#4404)
- pay: `pay` will now abort early if the destination is not reachable directly nor via routehints. (#4404)
- pay: `pay` was reporting in-flight parts as failed (#4404)
- pay: `pay` would crash on corrupt gossip store, which (if version was ever wrong) we'd corrupt again ([#4453])
- pyln: Fixed an error when calling `listfunds` with an older c-lightning version causing an error about an unknown `spent` parameter (#4417)
- Plugins: `dev-sendcustommsg` included the type and length prefix when sending a message. (#4413)
- Plugins: The `custommsg` hook no longer includes the internal type prefix and length prefix in its `payload` (#4394)
- db: Fixed an access to a NULL-field in the `channel_htlcs` table and resulting warning. (#4378)
- pay: Payments with an empty route (self-payment) are now aborted. (#4379)
- Protocol: always accept channel_updates from errors, even they'd otherwise be rejected as spam. (#4361)
- connectd: Occasional crash in connectd due to use-after-free (#4360)
- lightningd: JSON failures when –daemon is used without –log-file. (#4350)
- lightningd: don't assert if time goes backwards temporarily. ([#4449])

9.8.6 EXPERIMENTAL

These options are either enabled by explicit *experimental* config parameters, or building with `--enable-experimental-features`.

- lightningd: `experimental-dual-fund` runtime flag will enable dual-funded protocol on this node ([#4427])
- lightningd: `experimental-shutdown-wrong-funding` to allow remote nodes to close incorrectly opened channels. (#4421)
- JSON-RPC: close has a new `wrong_funding` option to try to close out unused channels where we messed up the funding tx. (#4421)
- JSON-RPC: Permit user-initiated aborting of in-progress opens. Only valid for not-yet-committed opens and RBF-attempts (#4424)
- JSON-RPC: `listpeers` now includes 'last_feerate', 'next_feerate', 'initial_feerate' and 'next_fee_step' for channels in state DUALOPEND_AWAITING_LOCKIN (#4399)

9.9 0.9.3 - 2021-01-20: Federal Qualitative Strengthening

This release named by Karol Hosiawa.

9.9.1 Added

- JSON-RPC: The `listfunds` method now includes spent outputs if the `spent` parameter is set to true. (#4296)
- JSON-RPC: `createinvoice` new low-level invoice creation API. (#4256)
- JSON-RPC: `invoice` now takes an optional `cltv` parameter. (#4320)
- JSON-RPC: `listinvoices` can now query for an invoice matching a `payment_hash` or a `bolt11` string, in addition to `label` (#4312)
- JSON-RPC: fundpsbt/utxopsbt have new param, `min_witness_utxo`, which sets a floor for the weight calculation of an added input (#4211)
- docs: `doc/BACKUP.md` describes how to back up your C-lightning node. (#4207)
- fee_base and fee_ppm to listpeers (#4247)
- hsmtool: password must now be entered on stdin. Password passed on the command line are discarded. (#4303)
- plugins: `start` command can now take plugin-specific parameters. (#4278)
- plugins: new "multi" field allows an option to be specified multiple times. (#4278)
- pyln-client: `fundpsbt/utxopsbt` now support `min_witness_weight` param (#4295)
- pyln: Added support for command notifications to LightningRpc via the `notify` context-manager. (#4311)
- pyln: Plugin methods can now report progress or status via the `Request.notify` function (#4311)
- pyln: plugins can now raise RpcException for finer control over error returns. (#4279)
- experimental-offers: enables fetch, payment and creation of (early draft) offers. (#4328)
- libplugin: init can return a non-NULL string to disable the plugin. (#4328)
- plugins: plugins can now disable themselves by returning `disable`, even if marked important. (#4328)
- experimental-onion-messages enables send, receive and relay of onion messages. (#4328)

9.9.2 Changed

- JSON-RPC: invalid UTF-8 strings now rejected. (#4227)
- bitcoin: The default network was changed from "testnet" to "mainnet", this only affects new nodes (#4277)
- cli: `lightning-cli` now performs better sanity checks on the JSON-RPC requests it sends. (#4259)
- hsmd: we now error at startup on invalid hsm_secret (#4307)
- hsmtool: all commands now error on invalid hsm_secret (#4307)
- hsmtool: the `encrypt` now asks you to confirm your password (#4307)
- lightningd: the `--encrypted-hsm` now asks you to confirm your password when first set (#4307)
- plugins: Multiple plugins can now register `db_write` hooks. (#4220)
- plugins: more than one plugin can now register `invoice_payment` hook. (#4226)
- pyln: Millisatoshi has new method, `to_whole_satoshi`; *rounds value up* to the nearest whole satoshi (#4295)
- pyln: `txprepare` no longer supports the deprecated `destination satoshi feerate utxos` call format. (#4259)

9.9.3 Deprecated

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

9.9.4 Removed

- plugins: options to `init` are no longer given as strings if they are bool or int types (deprecated in 0.8.2). (#4278)

9.9.5 Fixed

- JSON-RPC: The status of the shutdown meesages being exchanged is now displayed correctly. (#4263)
- JSONRPC: `setchannelfee` would fail an assertion if channel wasn't in normal state. (#4282)
- db: Fixed a performance regression during block sync, resulting in many more queries against the DB than necessary. (#4319)
- hsmtool: the `generatehsm` command now generates an appropriately-sized hsm_secret (#4304)
- keysend: Keysend now checks whether the destination supports keysend before attempting a payment. If not a more informative error is returned. (#4236)
- log: Do not terminate on the second received SIGHUP. (#4243)
- onchaind is much faster when unilaterally closing old channels. (#4250)
- onchaind uses much less memory on unilateral closes for old channels. (#4250)
- pay: Fixed an issue where waiting for the blockchain height to sync could time out. (#4317)
- pyln: parsing msat from a float string (#4237)
- hsmtool: fix a segfault on `dumponchaindescriptors` without network parameter (#4341)
- db: Speed up deletion of peer especially when there is a long history with that peer. (#4337)

9.9.6 Security

9.10 0.9.2 - 2020-11-20: Now with 0-of-N Multisig

This release named by Sergi Delgado Segura.

- Note: PSBTs now require bitcoind v0.20.1 or above *

9.10.1 Added

- JSON-RPC: Added 'state_changes' history to listpeers channels (4126)
- JSON-RPC: Added 'opener' and 'closer' to listpeers channels (4126)
- JSON-RPC: `close` now sends notifications for slow closes (if `allow-deprecated-apis`=false) (4046)
- JSON-RPC: `notifications` command to enable notifications. (4046)
- JSON-RPC: `multifundchannel` has a new optional argument, 'commitment_feerate', which can be used to differentiate between the funding feerate and the channel's initial commitment feerate (4139)
- JSON-RPC `fundchannel` now accepts an optional 'close_to' param, a bitcoin address that the channel funding should be sent to on close. Requires `opt_upfront_shutdownscript` (4132)
- Plugins: Channel closure resaon/cause to channel_state_changed notification (4126)
- Plugins: `htlc_accepted` hook can now return custom `failure_onion`. (4187)
- Plugins: hooks can now specify that they must be called 'before' or 'after' other plugins. (4168)
- hsmtool: a new command was added to hsmtool for dumping descriptors of the onchain wallet (4171)
- hsmtool: `hsm_secret` generation from a seed-phrase following BIP39. (4065)
- cli: print notifications and progress bars if commands provide them. (4046)
- pyln-client: pyln.client handles and can send progress notifications. (4046)
- pyln-client: Plugin method and hook requests prevent the plugin developer from accidentally setting the result multiple times, and will raise an exception detailing where the result was first set. (4094)
- pyln-client: Plugins have been integrated with the `logging` module for easier debugging and error reporting. (4101)
- pyln-proto: Added pure python implementation of the sphinx onion creation and processing functionality. (4056)
- libplugin: routines to send notification updates and progress. (4046)
- build: clang build now supports –enable-address-sanitizer . (4013)
- db: Added support for key-value DSNs for postgresql, allowing for a wider variety of configurations and environments. (4072)

9.10.2 Changed

- – Requires bitcoind v0.20.1 or above * (4179)
- Plugins: `pay` will now try disabled channels as a last resort. (4093)
- Protocol: mutual closing feerate reduced to "slow" to avoid overpaying. (4113)

- In-memory log buffer reduced from 100MB to 10MB (4087)

9.10.3 Deprecated

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

- cli: scripts should filter out '^# ' or use `-N none`, as commands will start returning notifications soon (4046)

9.10.4 Removed

- Protocol: Support for receiving full gossip from ancient LND nodes. (4184)
- JSON-RPC: `plugin stop` result with an empty ("") key (deprecated 0.8.1) (4049)
- JSON-RPC: The hook `rpc_command` returning `{"continue": true}` (deprecated 0.8.1) (4049)
- JSON-RPC: The hook `db_write` can no longer return `true` (deprecated in 0.8.1) (4049)
- JSON-RPC: `htlc_accepted` hook `per_hop_v0` object removed (deprecated 0.8.0) (4049)
- JSON-RPC: `listconfigs` duplicated "plugin" paths (deprecated 0.8.0) (4049)
- Plugin: Relative plugin paths are not relative to startup (deprecated v0.7.2.1) (4049)

9.10.5 Fixed

- Network: Fixed a race condition when us and a peer attempt to make channels to each other at nearly the same time. (4116)
- Protocol: fixed retransmission order of multiple new HTLCs (causing channel close with LND) (4124)
- Protocol: `signet` is now compatible with the final bitcoin-core version (4078)
- Crash: assertion fail at restart when source and destination channels of an HTLC are both onchain. (4122)
- We are now able to parse any amount string (XXXmsat, XX.XXXbtc, ..) we create. (4129)
- Some memory leaks in transaction and PSBT manipulate closed. (4071)
- openingd now uses the correct dust limit for determining the allowable floor for a channel open (affects fundee only) (4141)
- Plugin: Regression with SQL statement expansion that could result in invalid statements being passed to the `db_write` hook. (4090)
- build: no longer spuriously regenerates generated sources due to differences in `readdir`(3) sort order. (4053)
- db: Fixed a broken migration on postgres DBs that had really old channels. (4064)

9.10.6 Security

9.11 0.9.1 - 2020-09-15: The Antiguan BTC Maximalist Society

This release named by Jon Griffiths.

9.11.1 Added

- JSON-RPC: `multiwithdraw` command to batch multiple onchain sends in a single transaction. Note it shuffles inputs and outputs, does not use BIP69. (3812)
- JSON-RPC: `multifundchannel` command to fund multiple channels to different peers all in a single onchain transaction. (3763)
- JSON-RPC: `delpay` command to delete a payment once completed or failed. (3899)
- Plugins: `channel_state_changed` notification (4020)
- JSON-RPC: `listpays` can be used to query payments using the `payment_hash` (3888)
- JSON-RPC: `listpays` now includes the `payment_hash` (3888)
- JSON-RPC: `listpays` now includes the timestamp of the first part of the payment (3909)
- Build: New reproducible build system now uses docker: try it at home with `doc/REPRODUCIBLE.md`! (4021)
- Plugins: Proxy information now provided in `init.configuration`. (4010)
- Plugins: `openchannel_hook` is now chainable (3960)
- JSON-RPC: `listpeers` shows `features` list for each channel. (3963)
- JSON-RPC: `signpsbt` takes an optional `signonly` array to limit what inputs to sign. (3954)
- JSON-RPC: `utxopsbt` takes a new `locktime` parameter (3954)
- JSON-RPC: `fundpsbt` takes a new `locktime` parameter (3954)
- JSON-RPC: New low-level command `utxopsbt` to create PSBT from existing utxos. (3845)
- JSON-RPC: `listfunds` now has a `redeemscript` field for p2sh-wrapped outputs. (3844)
- JSON-RPC: `fundchannel` has new `outnum` field indicating which output of the transaction funds the channel. (3844)
- pyln-client: commands and options can now mark themselves deprecated. (3883)
- Plugins: can now mark their options and commands deprecated. (3883)
- plugins: `getmanifest` may now include "allow-deprecated-apis" boolean flag. (3883)
- JSON-RPC: `listpays` now lists the `destination` if it was provided (e.g., via the `pay` plugin or `keysend` plugin) (3888)
- config: New option `--important-plugin` loads a plugin is so important that if it dies, `lightningd` will exit rather than continue. You can still `--disable-plugin` it, however, which trumps `--important-plugin` and it will not be started at all. (3890)
- Plugins: We now explicitly check at startup that our default Bitcoin backend (bitcoind) does relay transactions. (3889)
- Plugins: We now explicitly check at startup the version of our default Bitcoin backend (bitcoind). (3889)

9.11.2 Changed

- Build: we no longer require extra Python modules to build. (3994)
- Build: SQLite3 is no longer a hard build requirement. C-Lightning can now be built to support only the PostgreSQL back-end. (3999)

- gossipd: The `gossipd` is now a lot quieter, and will log only when a message changed our network topology. (3981)
- Protocol: We now make MPP-aware routehints in invoices. (3913)
- onchaind: We now scorch the earth on theft attempts, RBFing up our penalty transaction as blocks arrive without a penalty transaction getting confirmed. (3870)
- Protocol: `fundchannel` now shuffles inputs and outputs, and no longer follows BIP69. (3769)
- JSON-RPC: `withdraw` now randomizes input and output order, not BIP69. (3867)
- JSON-RPC: `txprepare` reservations stay across restarts: use `fundpsbt/reservepsbt/unreservepsbt` (3867)
- config: `min-capacity-sat` is now stricter about checking usable capacity of channels. (3969)
- Protocol: Ignore (and log as "unusual") repeated `WIRE_CHANNEL_REESTABLISH` messages, to be compatible with buggy peer software that sometimes does this. (3964)
- contrib: startup_regtest.sh `startup_ln` now takes a number of nodes to create as a parameter (3992)
- JSON-RPC: `invoice` no longer accepts zero amounts (did you mean "any"?) (3974)
- Protocol: channels now pruned after two weeks unless both peers refresh it (see lightning-rfc#767) (3959)
- Protocol: bolt11 invoices always include CLTV fields (see lightning-rfc#785) (3959)
- config: the default CLTV expiry is now 34 blocks, and final expiry 18 blocks as per new BOLT recommendations. (3959)
- Plugins: Builtin plugins are now marked as important, and if they crash, will cause C-lightning to stop as well. (3890)
- Protocol: Funding timeout is now based on the header count reported by the bitcoin backend instead of our current blockheight which might be lower. (3897)
- JSON-RPC: `delinvoice` will now report specific error codes: 905 for failing to find the invoice, 906 for the invoice status not matching the parameter. (3853)

9.11.3 Deprecated

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

- Plugins: `bcli` replacements should note that `sendrawtransaction` now has a second required Boolean argument, `allowhighfees`, which if `true`, means ignore any fee limits and just broadcast the transaction. (3870)
- JSON-RPC: `listsendpays` will no longer add `null` if we don't know the `amount_msat` for a payment. (3883)
- Plugins: `getmanifest` without any parameters; plugins should accept any parameters for future use. (3883)

9.11.4 Removed

- JSON-RPC: txprepare `destination satoshi` argument form removed (deprecated v0.7.3) (3867)

9.11.5 Fixed

- Plugins: `pay presplit` modifier now supports large payments without exhausting the available HTLCs. (3986)
- Plugins: `pay` corrects a case where we put the sub-payment value instead of the *total* value in the `total_msat` field of a multi-part payment. (3914)
- Plugins: `pay` is less aggressive with forgetting routehints. (3914)
- Plugins: `pay` no longer ignores routehints if the payment exceeds 10,000 satoshi. This is particularly bad if the payee is only reachable via routehints in an invoice. (3908)
- Plugins: `pay` limits the number of splits if the payee seems to have a low number of channels that can enter it, given the max-concurrent-htlcs limit. (3936)
- Plugins: `pay` will now make reliable multi-part payments to nodes it doesn't have a node_announcement for. (4035)
- JSON-RPC: significant speedups for plugins which create large JSON replies (e.g. listpays on large nodes). (3957)
- doc: Many missing manual pages were completed (3938)
- Build: Fixed compile error on macos (4019)
- pyln: Fixed HTLCs hanging indefinitely if the hook function raises an exception. A safe fallback result is now returned instead. (4031)
- Protocol: We now hang up if peer doesn't respond to init message after 60 seconds. (4039)
- elementsd: PSBTs include correct witness_utxo struct for elements transactions (4033)
- cli: fixed crash with `listconfigs` in `-H` mode (4012)
- Plugins: `bcli` significant speedups for block synchronization (3985)
- Build: On systems with multiple installed versions of the PostgreSQL client library, C-Lightning might link against the wrong version or fail to find the library entirely. `./configure` now uses `pg_config` to locate the library. (3995)
- Build: On some operating systems the postgresql library would not get picked up. `./configure` now uses `pg_config` to locate the headers. (3991)
- libplugin: significant speedups for reading large JSON replies (e.g. calling listsendpays on large nodes, or listchannels / listnodes). (3957)

9.11.6 Security

9.12 0.9.0 - 2020-07-31: "Rat Poison Squared on Steroids"

This release was named by Sebastian Falbesoner.

9.12.1 Added

- plugin: `pay` was rewritten to use the new payment flow. See `legacypay` for old version (3809)
- plugin: `pay` will split payments that are failing due to their size into smaller parts, if recipient supports the `basic_mpp` option (3809)

- plugin: `pay` will split large payments into parts of approximately 10k sat if the recipient supports the `basic_mpp` option (3809)
- plugin: The pay plugin has a new `--disable-mpp` flag that allows opting out of the above two multi-part payment addition. (3809)
- JSON-RPC: new low-level coin selection `fundpsbt` routine. (3825)
- JSON-RPC: The `pay` command now uses the new payment flow, the new `legacypay` command can be used to issue payment with the legacy code if required. (3826)
- JSON-RPC: The `keysend` command allows sending to a node without requiring an invoice first. (3792)
- JSON-RPC: `listfunds` now has a 'scriptpubkey' field. (3821)
- docker: Docker build now includes `LIGHTNINGD_NETWORK` ENV variable which defaults to "bitcoin". An user can override this (e.g. by `-e` option in `docker run`) to run docker container in regtest or testnet or any valid argument to `--network`. (3813)
- cli: We now install `lightning-hsmtool` for your `hsm_secret` needs. (3802)
- JSON-RPC: new call `signpsbt` which will add the wallet's signatures to a provided psbt (3775)
- JSON-RPC: new call `sendpsbt` which will finalize and send a signed PSBT (3775)
- JSON-RPC: Adds two new rpc methods, `reserveinputs` and `unreserveinputs`, which allow for reserving or unreserving wallet UTXOs (3775)
- Python: `pyln.spec.bolt{1,2,4,7}` packages providing python versions of the spec text and defined messages. (3777)
- pyln: new module `pyln.proto.message.bolts` (3733)
- cli: New `--flat` mode for easy grepping of `lightning-cli` output. (3722)
- plugins: new notification type, `coin_movement`, which tracks all fund movements for a node (3614)
- plugin: Added a new `commitment_revocation` hook that provides the plugin with penalty transactions for all revoked transactions, e.g., to push them to a watchtower. (3659)
- JSON-API: `listchannels` now shows channel `features`. (3685)
- plugin: New `invoice_creation` plugin event (3658)
- docs: Install documentation now has information about building for Alpine linux (3660)
- plugin: Plugins can opt out of having an RPC connection automatically initialized on startup. (3857)
- JSON-RPC: `sendonion` has a new optional `bolt11` argument for when it's used to pay an invoice. (3878)
- JSON-RPC: `sendonion` has a new optional `msatoshi` that is used to annotate the payment with the amount received by the destination. (3878)

9.12.2 Changed

- JSON-RPC: `fundchannel_cancel` no longer requires its undocumented `channel_id` argument after `fundchannel_complete`. (3787)
- JSON-RPC: `fundchannel_cancel` will now succeed even when executed while a `fundchannel_complete` is ongoing; in that case, it will be considered as cancelling the funding *after* the `fundchannel_complete` succeeds. (3778)
- JSON-RPC: `listfunds` 'outputs' now includes reserved outputs, designated as 'reserved' = true (3764)

- JSON-RPC: `txprepare` now prepares transactions whose `nLockTime` is set to the tip blockheight, instead of using 0. `fundchannel` will use `nLockTime` set to the tip blockheight as well. (3797)
- build: default compile output is prettier and much less verbose (3686)
- config: the `plugin-disable` option now works even if specified before the plugin is found. (3679)
- plugins: The `autoclean` plugin is no longer dynamic (you cannot manage it with the `plugin` RPC command anymore). (3788)
- plugin: The `paystatus` output changed as a result of the payment flow rework (3809)

9.12.3 Deprecated

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

- JSON-RPC: the `legacypay` method from the pay plugin will be removed after `pay` proves stable (3809)

9.12.4 Removed

- protocol: support for optional fields of the reestablish message are now compulsory. (3782)

9.12.5 Fixed

- JSON-RPC: Reject some bad JSON at parsing. (3761)
- JSON-RPC: The `feerate` parameters now correctly handle the standardness minimum when passed as `perkb`. (3772)
- JSON-RPC: `listtransactions` now displays all txids as little endian (3741)
- JSON-RPC: `pay` now respects maxfeepercent, even for tiny amounts. (3693)
- JSON-RPC: `withdraw` and `txprepare` `feerate` can be a JSON number. (3821)
- bitcoin: `lightningd` now always exits if the Bitcoin backend failed unexpectedly. (3675)
- cli: Bash completion on `lightning-cli` now works again (3719)
- config: we now take the `--commit-fee` parameter into account. (3732)
- db: Fixed a failing assertion if we reconnect to a peer that we had a channel with before, and then attempt to insert the peer into the DB twice. (3801)
- hsmtool: Make the password argument optional for `guesstoremote` and `dumpcommitments` sub-commands, as shown in our documentation and help text. (3822)
- macOS: Build for macOS Catalina / Apple clang v11.0.3 fixed (3756)
- protocol: Fixed a deviation from BOLT#2: if both nodes advertised `option_upfront_shutdown_script` feature: MUST include … a zero-length `shutdown_scriptpubkey`. (3816)
- wumbo: negotiate successfully with Eclair nodes. (3712)
- plugin: `bcli` no longer logs a harmless warning about being unable to connect to the JSON-RPC interface. (3857)

9.12.6 Security

9.13 0.8.2 - 2020-04-30: “A Scalable Ethereum Blockchain”

This release was named by @arowser.

9.13.1 Added

- pay: The `keysend` plugin implements the ability to receive spontaneous payments (keysend) (3611)
- Plugin: the Bitcoin backend plugin API is now final. (3620)
- Plugin: `htlc_accepted` hook can now offer a replacement onion `payload`. (3611)
- Plugin: `feature_set` object added to `init` (3612)
- Plugin: ‘flag’-type option now available. (3586)
- JSON API: New `getsharedsecret` command, which lets you compute a shared secret with this node knowing only a public point. This implements the BOLT standard of hashing the ECDH point, and is incompatible with ECIES. (3490)
- JSON API: `large-channels` option to negotiate opening larger channels. (3612)
- JSON API: New optional parameter to the `close` command to control the closing transaction fee negotiation back off step (3390)
- JSON API: `connect` returns `features` of the connected peer on success. (3612)
- JSON API: `listpeers` now has `receivable_msat` (3572)
- JSON API: The fields “opening”, “mutual_close”, “unilateral_close”, “delayed_to_us”, “htlc_resolution” and “penalty” have been added to the `feerates` command. (3570)
- JSON API: “htlc_timeout_satoshis” and “htlc_success_satoshis” fields have been added to the `feerates` command. (3570)
- pyln now sends proper error on bad calls to plugin methods (3640)
- devtools: The `onion` tool can now generate, compress and decompress onions for rendez-vous routing (3557)
- doc: An FAQ was added, accessible at https://lightning.readthedocs.io/FAQ.html (3551)

9.13.2 Changed

- We now use a higher feerate for resolving onchain HTLCs and for penalty transactions (3592)
- We now announce multiple addresses of the same type, if given. (3609)
- pay: Improved the performance of the `pay`-plugin by limiting the `listchannels` when computing the shadow route. (3617)
- JSON API: `invoice exposeprivatechannels` now includes explicitly named channels even if they seem like dead-ends. (3633)
- Added workaround for lnd rejecting our commitment_signed when we send an update_fee after channel confirmed. (3634)
- We now batch the requests for fee estimation to our Bitcoin backend. (3570)
- We now get more fine-grained fee estimation from our Bitcoin backend. (3570)

- Forwarding messages is now much faster (less inter-daemon traffic) (3547)
- dependencies: We no longer depend on python2 which has reached end-of-life (3552)

9.13.3 Deprecated

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

- JSON API: `fundchannel_start satoshi` field really deprecated now (use `amount`). (3603)
- JSON API: The "urgent", "slow", and "normal" field of the `feerates` command are now deprecated. (3570)
- JSON API: Removed double wrapping of `rpc_command` payload in `rpc_command` JSON field. (3560)
- Plugins: htlc_accepted_hook "failure_code" only handles simple cases now, use "failure_message". (3472)
- Plugins: invoice_payment_hook "failure_code" only handles simple cases now, use "failure_message". (3472)

9.13.4 Removed

- JSON API: `listnodes globalfeatures` output (`features` since in 0.7.3). (3603)
- JSON API: `listpeers localfeatures` and `globalfeatures` output (`features` since in 0.7.3). (3603)
- JSON API: `peer_connected` hook `localfeatures` and `globalfeatures` output (`features` since in 0.7.3). (3603)
- JSON API: `fundchannel` and `fundchannel_start satoshi` parameter removed (renamed to `amount` in 0.7.3). (3603)
- JSON API: `close force` parameter removed (deprecated in 0.7.2.1) (3603)
- JSON API: `sendpay description` parameter removed (renamed to `label` in 0.7.0). (3603)

9.13.5 Fixed

- Plugins: A crashing plugin will no longer cause a hook call to be delayed indefinitely (3539)
- Plugins: setting an 'init' feature bit allows us to accept it from peers. (3609)
- Plugins: if an option has a type int or bool, return the option as that type to the plugin's init (3582)
- Plugins: Plugins no longer linger indefinitely if their process terminates (3539)
- JSON API: Pending RPC method calls are now terminated if the handling plugin exits prematurely. (3639)
- JSON API: `fundchannel_start` returns `amount` even when deprecated APIs are enabled. (3603)
- JSON API: Passing 0 as minconf to withdraw allows you to use unconfirmed transaction outputs, even if explicitly passed as the `utxos` parameter (3593)
- JSON API: `txprepare` doesn't crash lightningd anymore if you pass unconfirmed utxos (3534)
- invoice: The invoice parser assumed that an amount without a multiplier was denominated in msatoshi instead of bitcoins. (3636)
- pay: The `pay`-plugin was generating non-contiguous shadow routes (3617)
- `pay` would crash on expired waits with tried routes (3630)
- `pay` would crash when attempting to find cheaper route with exemptfee (3630)

- Multiple definition of chainparams on Fedora (or other really recent gcc) (3631)
- bcli now handles 0msat outputs in gettxout. (3605)
- Fix assertion on reconnect if we fail to run openingd. (3604)
- Use lightning-rfc #740 feespike margin factor of 2 (3589)
- Always broadcast the latest close transaction at the end of the close fee negotiation, instead of sometimes broadcasting the peer's initial closing proposal. (3556)

9.13.6 Security

9.14 0.8.1 - 2020-02-12: "Channel to the Moon"

This release named by Vasil Dimov @vasild.

9.14.1 Added

- Plugin: pluggable backends for Bitcoin data queries, default still bitcoind (using bitcoin-cli). (3488)
- Plugin: Plugins can now signal support for experimental protocol extensions by registering featurebits for `node_announcements`, the connection handshake, and for invoices. For now this is limited to non-dynamic plugins only (3477)
- Plugin: 'plugin start' now restores initial umask before spawning the plugin process (3375)
- JSON API: `fundchannel` and `fundchannel_start` can now accept an optional parameter, `push_msat`, which will gift that amount of satoshis to the peer at channel open. (3369)
- JSON API: `waitanyinvoice` now supports a `timeout` parameter, which when set will cause the command to fail if unpaid after `timeout` seconds (can be 0). (3449)
- Config: `--rpc-file-mode` sets permissions on the JSON-RPC socket. (3437)
- Config: `--subdaemon` allows alternate subdaemons. (3372)
- lightningd: Optimistic locking prevents instances from running concurrently against the same database, providing linear consistency to changes. (3358)
- hsmd: Added fields to hsm_sign_remote_commitment_tx to allow complete validation by signing daemon. (3363)
- Wallet: withdrawal transactions now sets nlocktime to the current tip. (3465)
- elements: Added support for the dynafed block header format and elementsd >=0.18.1 (3440)

9.14.2 Changed

- JSON API: The hooks `db_write`, `invoice_payment`, and `rpc_command` now accept `{ "result": "continue" }` to mean "do default action". (3475)
- Plugin: Multiple plugins can now register for the htlc_accepted hook. (3489)
- JSON API: `listforwards` now shows `out_channel` even if we couldn't forward.
- JSON API: `funchannel_cancel`: only the opener of a fundchannel can cancel the channel open (3336)

- JSON API: `sendpay` optional `msatoshi` param for non-MPP (if set), must be the exact amount sent to the final recipient. (3470)
- JSON API: `waitinvoice` now returns error code 903 to designate that the invoice expired during wait, instead of the previous -2 (3441)
- JSON_API: The `connect` command now returns its own error codes instead of a generic -1. (3397)
- Plugin: `notify_sendpay_success` and `notify_sendpay_failure` are now always called, even if there is no command waiting on the result. (3405)
- hsmtool: `hsmtool` now creates its backup copy in the same directory as the original `hsm_secret` file. (3409)
- JSON API: `invoice` `exposeprivatechannels` can specify exact channel candidates. (3351)
- JSON API: `db_write` new field `data_version` which contains a numeric transaction counter. (3358)
- JSON API: `plugin stop` result is now accessible using the `result` key instead of the empty (") key. (3374)
- lightning-cli: specifying `--rpc-file` (without `--network`) has been restored. (3353)

9.14.3 Deprecated

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

- JSON API: The hook `db_write` returning `true`: use `{ "result": "continue" }`. (3475)
- JSON API: The hook `invoice_payment` returning `{}`: use `{ "result": "continue" }`. (3475)
- JSON API: The hook `rpc_command` returning `{"continue": true}`: use `{ "result": "continue" }`. (3475)
- JSON API: `plugin stop` result with an empty ("") key: use "result". (3374)

9.14.4 Removed

- Plugin: Relative plugin paths are not relative to startup (deprecated v0.7.2.1) (3471)
- JSON API: Dummy fields in listforwards (deprecated v0.7.2.1) (3471)

9.14.5 Fixed

- Doc: Corrected and expanded `lightning-listpeers.7` documentation. (3497)
- Doc: Fixed factual errors in `lightning-listchannels.7` documentation. (3494)
- Protocol: Corner case where channel could become unusable (https://github.com/lightningnetwork/lightning-rfc/issues/728) (3500)
- Plugins: Dynamic C plugins can now be managed when lightningd is up (3480)
- Doc: `connect`: clarified failure problems and usage. (3459)
- Doc: `fundchannel`: clarify that we automatically `connects` if your node knows how. (3459)
- Protocol: Now correctly reject "fees" paid when we're the final hop (lightning-rfc#711) (3474)
- JSON API: `txprepare` no longer crashes when more than two outputs are specified (3384)
- Pyln: now includes the "jsonrpc" field to jsonrpc2 requests (3442)

- Plugin: `pay` now detects a previously non-permanent error (`final_cltv_too_soon`) that has been merged into a permanent error (`incorrect_or_unknown_payment_details`), and retries. (3376)
- JSON API: The arguments for `createonion` are now checked to ensure they fit in the onion packet. (3404)
- TOR: We don't send any further request if the return code of connect is not zero or error. (3408)
- Build: Developer mode compilation on FreeBSD. (3344)
- Protocol: We now reject invoices which ask for sub-millisatoshi amounts (3481)

9.14.6 Security

9.15 0.8.0 - 2019-12-16: "Blockchain Good, Orange Coin Bad"

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

9.15.1 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)

9.15.2 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)

9.15.3 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)

9.15.4 Removed

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

9.15.5 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)

9.15.6 Security

9.16 0.7.3 - 2019-10-18: "Bitcoin's Proof of Stake"

This release was named by @trueptolemy.

9.16.1 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

9.16.2 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.

9.16.3 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`).

9.16.4 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).

9.16.5 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).

9.16.6 Security

9.17 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)

9.17.1 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

9.17.2 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.

9.17.3 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`.

9.17.4 Removed

9.17.5 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.

9.17.6 Security

9.18 0.7.1 - 2019-06-29: "The Unfailing Twitter Consensus Algorithm"

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

9.18.1 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`.

9.18.2 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.

9.18.3 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.

9.18.4 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.

9.18.5 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.

9.18.6 Security

- Fixes CVE-2019-12998 (Full Disclosure).

9.19 0.7.0 - 2019-02-28: "Actually an Altcoin"

This release was named by Mark Beckwith @wythe.

9.19.1 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.

9.19.2 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.

9.19.3 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`)

9.19.4 Removed

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

9.19.5 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)

9.19.6 Security

9.20 0.6.3 - 2019-01-09: "The Smallblock Conspiracy"

This release was named by @molxyz and @ctrlbreak.

9.20.1 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.

9.20.2 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.

9.20.3 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.

9.20.4 Removed

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

9.20.5 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`

9.20.6 Security

9.21 0.6.2 - 2018-10-20: "The Consensus Loving Nasal Daemon"

This release was named by practicalswift.

9.21.1 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.

9.21.2 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.

9.21.3 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.

9.21.4 Removed

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

9.21.5 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.

9.21.6 Security

9.22 0.6.1 - 2018-09-11: "Principled Opposition To Segwit"

This release was named by ZmnSCPxj.

9.22.1 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.

9.22.2 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.

9.22.3 Deprecated

9.22.4 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.

9.22.5 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)

9.22.6 Security

9.23 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.

9.23.1 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`.

9.24 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 10

lightning-addgossip – Command for injecting a gossip message (low-level)

10.1 SYNOPSIS

addgossip *message*

10.2 DESCRIPTION

The **addgossip** RPC command injects a hex-encoded gossip message into the gossip daemon. It may return an error if it is malformed, or may update its internal state using the gossip message.

Note that currently some paths will still silently reject the gossip: it is best effort.

This is particularly used by plugins which may receive channel_update messages within error replies.

10.3 RETURN VALUE

On success, an empty object is returned.

10.4 AUTHOR

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

10.5 SEE ALSO

lightning-pay(7)

10.6 RESOURCES

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

CHAPTER 11

lightning-autoclean-status – Examine auto-delete of old invoices/payments/forwards

11.1 SYNOPSIS

autoclean-status [*subsystem*]

11.2 DESCRIPTION

The **autoclean-status** RPC command tells you about the status of the autclean plugin, optionally for only one subsystem.

The subsystems currently supported are:

- `failedforwards`: routed payments which did not succeed (`failed` or `local_failed` in listforwards `status`).
- `succeededforwards`: routed payments which succeeded (`settled` in listforwards `status`).
- `failedpays`: payment attempts which did not succeed (`failed` in listpays `status`).
- `succededpays`: payment attempts which succeeded (`complete` in listpays `status`).
- `expiredinvoices`: invoices which were not paid (and cannot be) (`expired` in listinvoices `status`).
- `paidinvoices`: invoices which were paid (`paid` in listinvoices ‘status).

11.3 RETURN VALUE

Note that the ages parameters are set by various `autoclean-...-age` parameters in your configuration: see lightningd-config(5).

On success, an object containing **autoclean** is returned. It is an object containing:

- **succeededforwards** (object, optional):
  - **enabled** (boolean): whether autocleaning is enabled for successful listforwards
  - **cleaned** (u64): total number of deletions done (ever)

  If **enabled** is *true*:

  - **age** (u64): age (in seconds) to delete successful listforwards
- **failedforwards** (object, optional):
  - **enabled** (boolean): whether autocleaning is enabled for failed listforwards
  - **cleaned** (u64): total number of deletions done (ever)

  If **enabled** is *true*:

  - **age** (u64): age (in seconds) to delete failed listforwards
- **succeededpays** (object, optional):
  - **enabled** (boolean): whether autocleaning is enabled for successful listpays/listsendpays
  - **cleaned** (u64): total number of deletions done (ever)

  If **enabled** is *true*:

  - **age** (u64): age (in seconds) to delete successful listpays/listsendpays
- **failedpays** (object, optional):
  - **enabled** (boolean): whether autocleaning is enabled for failed listpays/listsendpays
  - **cleaned** (u64): total number of deletions done (ever)

  If **enabled** is *true*:

  - **age** (u64): age (in seconds) to delete failed listpays/listsendpays
- **paidinvoices** (object, optional):
  - **enabled** (boolean): whether autocleaning is enabled for paid listinvoices
  - **cleaned** (u64): total number of deletions done (ever)

  If **enabled** is *true*:

  - **age** (u64): age (in seconds) to paid listinvoices
- **expiredinvoices** (object, optional):
  - **enabled** (boolean): whether autocleaning is enabled for expired (unpaid) listinvoices
  - **cleaned** (u64): total number of deletions done (ever)

  If **enabled** is *true*:

  - **age** (u64): age (in seconds) to expired listinvoices

11.4 AUTHOR

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

11.5 SEE ALSO

lightningd-config(5), lightning-listinvoices(7), lightning-listpays(7), lightning-listforwards(7).

11.6 RESOURCES

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

CHAPTER 12

lightning-batching – Command to allow database batching.

12.1 SYNOPSIS

batching *enable*

12.2 DESCRIPTION

The **batching** RPC command allows (but does not guarantee!) database commitments to be deferred when multiple commands are issued on this RPC connection. This is only useful if many commands are being given at once, in which case it can offer a performance improvement (the cost being that if there is a crash, it's unclear how many of the commands will have been persisted).

enable is *true* to enable batching, *false* to disable it (the default).

12.3 EXAMPLE JSON REQUEST

```
{
  "id": 82,
  "method": "batching",
  "params": {
    "enable": true
  }
}
```

12.4 RETURN VALUE

On success, an empty object is returned.

On failure, one of the following error codes may be returned:

- -32602: Error in given parameters.

12.5 AUTHOR

Rusty Russell <rusty@blockstream.com> wrote the initial version of this man page.

12.6 RESOURCES

Main web site: https://github.com/ElementsProject/lightning *comment*: # (SHA256STAMP:326e5801f65998e13e909d8b682e9fbc9824f3a43aa7da1d76b871882e52f293)

CHAPTER 13

lightning-bkpr-channelsapy – Command to list stats on channel earnings

13.1 SYNOPSIS

bkpr-channelsapy [*start_time*] [*end_time*]

13.2 DESCRIPTION

The **bkpr-channelsapy** RPC command lists stats on routing income, leasing income, and various calculated APYs for channel routed funds.

The **start_time** is a UNIX timestamp (in seconds) that filters events after the provided timestamp. Defaults to zero.

The **end_time** is a UNIX timestamp (in seconds) that filters events up to and at the provided timestamp. Defaults to max-int.

13.3 RETURN VALUE

On success, an object containing **channels_apy** is returned. It is an array of objects, where each object contains:

- **account** (string): The account name. If the account is a channel, the channel_id. The 'net' entry is the rollup of all channel accounts
- **routed_out_msat** (msat): Sats routed (outbound)
- **routed_in_msat** (msat): Sats routed (inbound)
- **lease_fee_paid_msat** (msat): Sats paid for leasing inbound (liquidity ads)
- **lease_fee_earned_msat** (msat): Sats earned for leasing outbound (liquidity ads)
- **pushed_out_msat** (msat): Sats pushed to peer at open
- **pushed_in_msat** (msat): Sats pushed in from peer at open

- **our_start_balance_msat** (msat): Starting balance in channel at funding. Note that if our start ballance is zero, any _initial field will be omitted (can't divide by zero)
- **channel_start_balance_msat** (msat): Total starting balance at funding
- **fees_out_msat** (msat): Fees earned on routed outbound
- **utilization_out** (string): Sats routed outbound / total start balance
- **utilization_in** (string): Sats routed inbound / total start balance
- **apy_out** (string): Fees earned on outbound routed payments / total start balance for the length of time this channel has been open amortized to a year (APY)
- **apy_in** (string): Fees earned on inbound routed payments / total start balance for the length of time this channel has been open amortized to a year (APY)
- **apy_total** (string): Total fees earned on routed payments / total start balance for the length of time this channel has been open amortized to a year (APY)
- **fees_in_msat** (msat, optional): Fees earned on routed inbound
- **utilization_out_initial** (string, optional): Sats routed outbound / our start balance
- **utilization_in_initial** (string, optional): Sats routed inbound / our start balance
- **apy_out_initial** (string, optional): Fees earned on outbound routed payments / our start balance for the length of time this channel has been open amortized to a year (APY)
- **apy_in_initial** (string, optional): Fees earned on inbound routed payments / our start balance for the length of time this channel has been open amortized to a year (APY)
- **apy_total_initial** (string, optional): Total fees earned on routed payments / our start balance for the length of time this channel has been open amortized to a year (APY)
- **apy_lease** (string, optional): Lease fees earned over total amount leased for the lease term, amortized to a year (APY). Only appears if channel was leased out by us

13.4 AUTHOR

niftynei niftynei@gmail.com is mainly responsible.

13.5 SEE ALSO

lightning-bkpr-listincome(7), lightning-bkpr-listfunds(7), lightning-bkpr-listaccountevents(7), lightning-bkpr-dumpincomecsv(7), lightning-listpeers(7).

13.6 RESOURCES

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

CHAPTER 14

lightning-bkpr-dumpincomecsv – Command to emit a CSV of income events

14.1 SYNOPSIS

bkpr-dumpincomecsv *csv_format* [*csv_file*] [*consolidate_fees*] [*start_time*] [*end_time*]

14.2 DESCRIPTION

The **bkpr-dumpincomcsv** RPC command writes a CSV file to disk at *csv_file* location. This is a formatted output of the **listincome** RPC command.

csv_format is which CSV format to use. See RETURN VALUE for options.

csv_file is the on-disk destination of the generated CSV file.

If **consolidate_fees** is true, we emit a single, consolidated event for any onchain-fees for a txid and account. Otherwise, events for every update to the onchain fee calculation for this account and txid will be printed. Defaults to true. Note that this means that the events emitted are non-stable, i.e. calling **dumpincomecsv** twice may result in different onchain fee events being emitted, depending on how much information we've logged for that transaction.

The **start_time** is a UNIX timestamp (in seconds) that filters events after the provided timestamp. Defaults to zero.

The **end_time** is a UNIX timestamp (in seconds) that filters events up to and at the provided timestamp. Defaults to max-int.

14.3 RETURN VALUE

On success, an object is returned, containing:

- **csv_file** (string): File that the csv was generated to
- **csv_format** (string): Format to print csv as (one of "cointracker", "koinly", "harmony", "quickbooks")

14.4 AUTHOR

niftynei niftynei@gmail.com is mainly responsible.

14.5 SEE ALSO

lightning-bkpr-listincome(7), lightning-bkpr-listfunds(7), lightning-bkpr-listaccountevents(7), lightning-bkpr-channelsapy(7), lightning-listpeers(7).

14.6 RESOURCES

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

CHAPTER 15

lightning-bkpr-inspect – Command to show onchain footprint of a channel

15.1 SYNOPSIS

bkpr-inspect *account*

15.2 DESCRIPTION

The **bkpr-inspect** RPC command lists all known on-chain transactions and associated events for the provided account. Useful for inspecting unilateral closes for a given channel account. Only valid for channel accounts.

15.3 RETURN VALUE

On success, an object containing **txs** is returned. It is an array of objects, where each object contains:

- **txid** (txid): transaction id
- **fees_paid_msat** (msat): Amount paid in sats for this tx
- **outputs** (array of objects):
  - **account** (string): Account this output affected
  - **outnum** (u32): Index of output
  - **output_value_msat** (msat): Value of the output
  - **currency** (string): human-readable bech32 part for this coin type
  - **credit_msat** (msat, optional): Amount credited to account
  - **debit_msat** (msat, optional): Amount debited from account
  - **originating_account** (string, optional): Account this output originated from

  - **output_tag** (string, optional): Description of output creation event
  - **spend_tag** (string, optional): Description of output spend event
  - **spending_txid** (txid, optional): Transaction this output was spent in
  - **payment_id** (hex, optional): lightning payment identifier. For an htlc, this will be the preimage.
- **blockheight** (u32, optional): Blockheight of transaction

15.4 AUTHOR

niftynei niftynei@gmail.com is mainly responsible.

15.5 SEE ALSO

lightning-listbalances(7), lightning-listfunds(7), lightning-listpeers(7).

15.6 RESOURCES

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

CHAPTER 16

lightning-bkpr-listaccountevents – Command for listing recorded bookkeeping events

16.1 SYNOPSIS

bkpr-listaccountevents [*account*]

16.2 DESCRIPTION

The **bkpr-listaccountevents** RPC command is a list of all bookkeeping events that have been recorded for this node.

If the optional parameter **account** is set, we only emit events for the specified account, if exists.

Note that the type **onchain_fees** that are emitted are of opposite credit/debit than as they appear in **listincome**, as **listincome** shows all events from the perspective of the node, whereas **listaccountevents** just dumps the event data as we've got it. Onchain fees are updated/recorded as we get more information about input and output spends – the total onchain fees that were recorded for a transaction for an account can be found by summing all onchain fee events and taking the difference between the **credit_msat** and **debit_msat** for these events. We do this so that successive calls to **listaccountevents** always produce the same list of events – no previously emitted event will be subsequently updated, rather we add a new event to the list.

16.3 RETURN VALUE

On success, an object containing **events** is returned. It is an array of objects, where each object contains:

- **account** (string): The account name. If the account is a channel, the channel_id
- **type** (string): Coin movement type (one of "onchain_fee", "chain", "channel")
- **tag** (string): Description of movement
- **credit_msat** (msat): Amount credited

- **debit_msat** (msat): Amount debited
- **currency** (string): human-readable bech32 part for this coin type
- **timestamp** (u32): Timestamp this event was recorded by the node. For consolidated events such as onchain_fees, the most recent timestamp

If **type** is "chain":

- **outpoint** (string): The txid:outnum for this event
- **blockheight** (u32): For chain events, blockheight this occured at
- **origin** (string, optional): The account this movement originated from
- **payment_id** (hex, optional): lightning payment identifier. For an htlc, this will be the preimage.
- **txid** (txid, optional): The txid of the transaction that created this event
- **description** (string, optional): The description of this event

If **type** is "onchain_fee":

- **txid** (txid): The txid of the transaction that created this event

If **type** is "channel":

- **fees_msat** (msat, optional): Amount paid in fees
- **is_rebalance** (boolean, optional): Is this payment part of a rebalance
- **payment_id** (hex, optional): lightning payment identifier. For an htlc, this will be the preimage.
- **part_id** (u32, optional): Counter for multi-part payments

16.4 AUTHOR

niftynei niftynei@gmail.com is mainly responsible.

16.5 SEE ALSO

lightning-bkpr-listincome(7), lightning-listfunds(7), lightning-bkpr-listbalances(7), lightning-bkpr-channelsapy(7).

16.6 RESOURCES

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

CHAPTER 17

lightning-bkpr-listbalances – Command for listing current channel + wallet balances

17.1 SYNOPSIS

bkpr-listbalances

17.2 DESCRIPTION

The **bkpr-listbalances** RPC command is a list of all current and historical account balances. An account is either the on-chain *wallet* or a channel balance. Any funds sent to an *external* account will not be accounted for here.

Note that any channel that was recorded will be listed. Closed channel balances will be 0msat.

17.3 RETURN VALUE

On success, an object containing **accounts** is returned. It is an array of objects, where each object contains:

- **account** (string): The account name. If the account is a channel, the channel_id
- **balances** (array of objects):
  - **balance_msat** (msat): Current account balance
  - **coin_type** (string): coin type, same as HRP for bech32

If **peer_id** is present:

- **peer_id** (pubkey): Node id for the peer this account is with
- **we_opened** (boolean): Did we initiate this account open (open the channel)
- **account_closed** (boolean):

- **account_resolved** (boolean): Has this channel been closed and all outputs resolved?
- **resolved_at_block** (u32, optional): Blockheight account resolved on chain

17.4 AUTHOR

niftynei niftynei@gmail.com is mainly responsible.

17.5 SEE ALSO

lightning-bkpr-listincome(7), lightning-listfunds(7), lightning-bkpr-listaccountevents(7), lightning-bkpr-channelsapy(7), lightning-listpeers(7).

17.6 RESOURCES

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

CHAPTER 18

lightning-bkpr-listincome – Command for listing all income impacting events

18.1 SYNOPSIS

bkpr-listincome [*consolidate_fees*] [*start_time*] [*end_time*]

18.2 DESCRIPTION

The **bkpr-listincome** RPC command is a list of all income impacting events that the bookkeeper plugin has recorded for this node.

If **consolidate_fees** is true, we emit a single, consolidated event for any onchain-fees for a txid and account. Otherwise, events for every update to the onchain fee calculation for this account and txid will be printed. Defaults to true. Note that this means that the events emitted are non-stable, i.e. calling **listincome** twice may result in different onchain fee events being emitted, depending on how much information we've logged for that transaction.

The **start_time** is a UNIX timestamp (in seconds) that filters events after the provided timestamp. Defaults to zero.

The **end_time** is a UNIX timestamp (in seconds) that filters events up to and at the provided timestamp. Defaults to max-int.

18.3 RETURN VALUE

On success, an object containing **income_events** is returned. It is an array of objects, where each object contains:

- **account** (string): The account name. If the account is a channel, the channel_id
- **tag** (string): Type of income event
- **credit_msat** (msat): Amount earned (income)
- **debit_msat** (msat): Amount spent (expenses)

- **currency** (string): human-readable bech32 part for this coin type
- **timestamp** (u32): Timestamp this event was recorded by the node. For consolidated events such as onchain_fees, the most recent timestamp
- **description** (string, optional): More information about this event. If a `invoice` type, typically the bolt11/bolt12 description
- **outpoint** (string, optional): The txid:outnum for this event, if applicable
- **txid** (txid, optional): The txid of the transaction that created this event, if applicable
- **payment_id** (hex, optional): lightning payment identifier. For an htlc, this will be the preimage.

18.4 AUTHOR

niftynei niftynei@gmail.com is mainly responsible.

18.5 SEE ALSO

lightning-bkpr-listaccountevents(7), lightning-listfunds(7), lightning-bkpr-listbalances(7).

18.6 RESOURCES

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

CHAPTER 19

lightning-check – Command for verifying parameters

19.1 SYNOPSIS

check *command_to_check* [*parameters*]

19.2 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.

19.3 RETURN VALUE

On success, an object is returned, containing:

- **command_to_check** (string): the *command_to_check* argument

19.4 AUTHOR

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

19.5 RESOURCES

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

CHAPTER 20

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

20.1 SYNOPSIS

checkmessage *message* *zbase* [*pubkey*]

20.2 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.

On failure, an error is returned and core lightning exit with the following error code:

- -32602: Parameter missed or malformed;
- 1301: *pubkey* not found in the graph.

20.3 RETURN VALUE

On success, an object is returned, containing:

- **verified** (boolean): whether the signature was valid (always *true*)
- **pubkey** (pubkey): the *pubkey* parameter, or the pubkey found by looking for known nodes

20.4 AUTHOR

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

20.5 SEE ALSO

lightning-signmessage(7)

20.6 RESOURCES

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

CHAPTER 21

lightning-cli – Control lightning daemon

21.1 SYNOPSIS

lightning-cli [*OPTIONS*] *command*

21.2 DESCRIPTION

lightning-cli sends commands to the lightning daemon.

21.3 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, or result contains a `format-hint` field).

- **–raw/-R**

  Return raw JSON directly as lightningd replies; this can be faster for large requests.

- **–human-readable/-H**

  Return result in human-readable output.

- **–flat/-F**

  Return JSON result in flattened one-per-line output, e.g. `{ "help": [ { "command": "check" } ] }` would become `help[0].command=check`. This is useful for simple scripts which want to find a specific output field without parsing JSON.

- **–notifications/-N**=*LEVEL*

  If *LEVEL* is 'none', then never print out notifications. Otherwise, print out notifications of *LEVEL* or above (one of `io`, `debug`, `info` (the default), `unusual` or `broken`: they are prefixed with `# `.

- **–filter/-l**=*JSON*

  This hands lightningd *JSON* as a filter, which controls what will be output, e.g. `'--filter={"help":[{"command":true}]}'`. See lightningd-rpc(7) for more details on how to specify filters.

- **–help/-h**

  Pretty-print summary of options to standard output and exit. The format can be changed using `-F`, `-R`, `-J`, `-H` etc.

- **–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.

21.4 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.

21.5 ARGUMENTS

Arguments may be provided positionally or using *key*=*value* after the command name, based on either **-o** or **-k** option. When using **-k** consider prefixing all arguments of the command with their respective keyword, this is to avoid having lightningd intrepret the position of an arguement.

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*. Arguments which begin with *{*, *[* or *"* are also considered raw JSON and are passed through. Other arguments are treated as strings.

Some commands have optional arguments. You may use *null* to skip optional arguments to provide later arguments, although this is not encouraged.

21.6 EXAMPLES

1. List commands:

- `lightning-cli help`

1. Fund a 10k sat channel using uncomfirmed outputs:

- `lightning-cli --keywords fundchannel id=028f...ae7d amount=10000sat minconf=0`

21.7 BUGS

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

21.8 AUTHOR

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

21.9 RESOURCES

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

21.10 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 22

lightning-close – Command for closing channels with direct peers

22.1 SYNOPSIS

close *id* [*unilateraltimeout*] [*destination*] [*fee_negotiation_step*] [*wrong_funding*] [*force_lease_closed*] [*feerange*]

22.2 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 bech32 type. If it isn't specified, the default is a Core Lightning wallet address. If the peer hasn't offered the `option_shutdown_anysegwit` feature, then taproot addresses (or other v1+ segwit) are not allowed. Tell your friends to upgrade!

The *fee_negotiation_step* parameter controls how closing fee negotiation is performed assuming the peer proposes a fee that is different than our estimate. (Note that modern peers use the quick-close protocol which does not allow negotiation: see *feerange* instead).

On every negotiation step we must give up some amount from our proposal towards the peer's proposal. This parameter can be an integer in which case it is interpreted as number of satoshis to step at a time. Or it can be an integer followed by "%" to designate a percentage of the interval to give up. A few examples, assuming the peer proposes a closing fee of 3000 satoshi and our estimate shows it must be 4000:

- "10": our next proposal will be 4000-10=3990.
- "10%": our next proposal will be 4000-(10% of (4000-3000))=3900.

- "1": our next proposal will be 3999. This is the most extreme case when we insist on our fee as much as possible.
- "100%": our next proposal will be 3000. This is the most relaxed case when we quickly accept the peer's proposal.

The default is "50%".

wrong_funding can only be specified if both sides have offered the "shutdown_wrong_funding" feature (enabled by the **experimental-shutdown-wrong-funding** option): it must be a transaction id followed by a colon then the output number. Instead of negotiating a shutdown to spend the expected funding transaction, the shutdown transaction will spend this output instead. This is only allowed if this peer opened the channel and the channel is unused: it can rescue openings which have been manually miscreated.

force_lease_closed if the channel has funds leased to the peer (option_will_fund), we prevent initiation of a mutual close unless this flag is passed in. Defaults to false.

feerange is an optional array [*min*, *max*], indicating the minimum and maximum feerates to offer: the peer will obey these if it supports the quick-close protocol. *slow* and *unilateral_close* are the defaults.

Rates are 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, or one of the names from lightning-feerates(7). Otherwise, they can be numbers 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*.

Note that the maximum fee will be capped at the final commitment transaction fee (unless the experimental anchor-outputs option is negotiated).

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.

22.3 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.

22.4 NOTIFICATIONS

Notifications may be returned indicating what is going on, especially if the peer is offline and we are waiting.

22.5 RETURN VALUE

On success, an object is returned, containing:

- **type** (string): Whether we successfully negotiated a mutual close, closed without them, or discarded not-yet-opened channel (one of "mutual", "unilateral", "unopened")

If **type** is "mutual" or "unilateral":

- **tx** (hex): the raw bitcoin transaction used to close the channel (if it was open)
- **txid** (txid): the transaction id of the *tx* field

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.

22.6 AUTHOR

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

22.7 SEE ALSO

lightning-disconnect(7), lightning-fundchannel(7), lightningd-config(5).

22.8 RESOURCES

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

CHAPTER 23

lightning-commando – Command to Send a Command to a Remote Peer

23.1 SYNOPSIS

commando *peer_id method* [*params*] [*rune*]

23.2 DESCRIPTION

The **commando** RPC command is a homage to bad 80s movies. It also sends a directly-connected *peer_id* a custom message, containing a request to run *method* (with an optional dictionary of *params*); generally the peer will only allow you to run a command if it has provided you with a *rune* which allows it.

23.3 RETURN VALUE

On success, the return depends on the *method* invoked.

On failure, one of the following error codes may be returned:

- -32600: Usually means peer is not connected
- 19535: the local commando plugin discovered an error.
- 19536: the remote commando plugin discovered an error.
- 19537: the remote commando plugin said we weren't authorized.

It can also fail if the peer does not respond, in which case it will simply hang awaiting a response.

23.4 AUTHOR

Rusty Russell <rusty@rustcorp.com.au> wrote the original Python commando.py plugin, the in-tree commando plugin, and this manual page.

Christian Decker came up with the name "commando", which almost excuses his previous adoption of the name "Eltoo".

23.5 SEE ALSO

lightning-commando-rune(7)

23.6 RESOURCES

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

CHAPTER 24

lightning-commando-rune – Command to Authorize Remote Peer Access

24.1 SYNOPSIS

commando-rune [*rune*] [*restrictions*]

24.2 DESCRIPTION

The **commando-rune** RPC command creates a base64 string called a *rune* which can be used to access commands on this node. Each *rune* contains a unique id (a number starting at 0), and can have restrictions inside it. Nobody can remove restrictions from a rune: if you try, the rune will be rejected. There is no limit on how many runes you can issue: the node doesn't store them, but simply decodes and checks them as they are received.

If *rune* is supplied, the restrictions are simple appended to that *rune* (it doesn't need to be a rune belonging to this node). If no *rune* is supplied, a new one is constructed, with a new unique id.

restrictions can be the string "readonly" (creates a rune which allows most *get* and *list* commands, and the *summary* command), or an array of restrictions.

Each restriction is an array of one or more alternatives, such as "method is listpeers", or "method is listpeers OR time is before 2023". Alternatives use a simple language to examine the command which is being run:

- time: the current UNIX time, e.g. "time<1656759180".
- id: the node_id of the peer, e.g. "id=024b9a1fa8e006f1e3937f65f66c408e6da8e1ca728ea43222a7381df1cc449605".
- method: the command being run, e.g. "method=withdraw".
- rate: the rate limit, per minute, e.g. "rate=60".
- pnum: the number of parameters. e.g. "pnum<2".
- pnameX: the parameter named X. e.g. "pnamedestination=1RustyRX2oai4EYYDpQGWvEL62BBGqN9T".
- parrN: the N'th parameter. e.g. "parr0=1RustyRX2oai4EYYDpQGWvEL62BBGqN9T".

24.3 RESTRICTION FORMAT

Restrictions are one or more alternatives. Each alternative is *name operator value*. The valid names are shown above. Note that if a value contains `\\`, it must be preceeded by another `\\` to form valid JSON:

- `=`: passes if equal ie. identical. e.g. `method=withdraw`
- `/`: not equals, e.g. `method/withdraw`
- `^`: starts with, e.g. `id^024b9a1fa8e006f1e3937f`
- `$`: ends with, e.g. `id$381df1cc449605`.
- `~`: contains, e.g. `id~006f1e3937f65f66c40`.
- `<`: is a decimal integer, and is less than. e.g. `time<1656759180`
- `>`: is a decimal integer, and is greater than. e.g. `time>1656759180`
- `{`: preceeds in alphabetical order (or matches but is shorter), e.g. `id{02ff`.
- `}`: follows in alphabetical order (or matches but is longer), e.g. `id}02ff`.
- `#`: a comment, ignored, e.g. `dumb example#`.
- `!`: only passes if the *name* does *not* exist. e.g. `pnamedestination!`. Every other operator except `#` fails if *name* does not exist!

24.4 EXAMPLES

This creates a fresh rune which can do anything:

```
$ lightning-cli commando-rune
{
   "rune": "KUhZzNlECC7pYsz3QVbF1TqjIUYi3oyESTI7n60hLMs9MA==",
   "unique_id": "0"
}
```

We can add restrictions to that rune, like so:

```
$ lightning-cli commando-rune rune=KUhZzNlECC7pYsz3QVbF1TqjIUYi3oyESTI7n60hLMs9MA==␣
↪restrictions=readonly
{
   "rune": "NbL7KkXcPQsVseJ9TdJNjJK2KsPjnt_q4cE_
↪wvc873I9MCZtZXRob2RebGlzdHxtZXRob2ReZ2V0fG1ldGhvZD1zdW1tYXJ5Jm1ldGhvZC9saXN0ZGF0YXN0b3Jl
↪",
   "unique_id": "0"
}
```

The “readonly” restriction is a short-cut for two restrictions:

1. `["method^list", "method^get", "method=summary"]`: You may call list, get or summary.
2. `["method/listdatastore"]`: But not listdatastore: that contains sensitive stuff!

We can do the same manually, like so:

```
$ lightning-cli commando-rune rune=KUhZzNlECC7pYsz3QVbF1TqjIUYi3oyESTI7n60hLMs9MA==␣
↪restrictions='[["method^list", "method^get", "method=summary"],["method/
↪listdatastore"]]'
{
   "rune": "NbL7KkXcPQsVseJ9TdJNjJK2KsPjnt_q4cE_
↪wvc873I9MCZtZXRob2RebGlzdHxtZXRob2ReZ2V0fG1ldGhvZD1zdW1tYXJ5Jm1ldGhvZC9saXN0ZGF0YXN0b3Jl
↪",
   "unique_id": "0"
}
```

Let's create a rune which lets a specific peer (024b9a1fa8e006f1e3937f65f66c408e6da8e1ca728ea43222a7381df1cc449605) run "listpeers" on themselves:

```
$ lightning-cli commando-rune restrictions='[[
↪"id=024b9a1fa8e006f1e3937f65f66c408e6da8e1ca728ea43222a7381df1cc449605"],[
↪"method=listpeers"],["pnum=1"],[
↪"pnameid=024b9a1fa8e006f1e3937f65f66c408e6da8e1ca728ea43222a7381df1cc449605",
↪"parr0=024b9a1fa8e006f1e3937f65f66c408e6da8e1ca728ea43222a7381df1cc449605"]]'
{
   "rune": "FE8GHiGVvxcFqCQcClVRRiNE_
↪XEeLYQzyG2jmqto4jM9MiZpZD0wMjRiOWExZmE4ZTAwNmYxZTM5MzdmNjVmNjZjNDA4ZTZkYThlMWNhNzI4ZWE0MzIyMmE3Mzgx
↪",
   "unique_id": "2"
}
```

This allows `listpeers` with 1 argument (`pnum=1`), which is either by name (`pnameid`), or position (`parr0`). We could shorten this in several ways: either allowing only positional or named parameters, or by testing the start of the parameters only. Here's an example which only checks the first 9 bytes of the `listpeers` parameter:

```
$ lightning-cli commando-rune restrictions='[[
↪"id=024b9a1fa8e006f1e3937f65f66c408e6da8e1ca728ea43222a7381df1cc449605"],[
↪"method=listpeers"],["pnum=1"],["pnameid^024b9a1fa8e006f1e393", "parr0^
↪024b9a1fa8e006f1e393"]'
 {
   "rune":
↪"fTQnfL05coEbiBO8SS0cvQwCcPLxE9c02pZCC6HRVEY9MyZpZD0wMjRiOWExZmE4ZTAwNmYxZTM5MzdmNjVmNjZjNDA4ZTZkY
↪",
   "unique_id": "3"
}
```

Before we give this to our peer, let's add two more restrictions: that it only be usable for 24 hours from now (`time<`), and that it can only be used twice a minute (`rate=2`). `date +%s` can give us the current time in seconds:

```
$ lightning-cli commando-rune␣
↪rune=fTQnfL05coEbiBO8SS0cvQwCcPLxE9c02pZCC6HRVEY9MyZpZD0wMjRiOWExZmE4ZTAwNmYxZTM5MzdmNjVmNjZjNDA4Z
↪restrictions='[["time<'$(($(date +%s) + 24*60*60))'","rate=2"]]'
{
   "rune": "tU-RLjMiDpY2U0o3W1oFowar36RFGpWloPbW9-
↪RuZdo9MyZpZD0wMjRiOWExZmE4ZTAwNmYxZTM5MzdmNjVmNjZjNDA4ZTZkYThlMWNhNzI4ZWE0MzIyMmE3MzgxZGYxY2M0NDk2N
↪",
   "unique_id": "3"
}
```

You can also use lightning-decode(7) to examine runes you have been given:

```
$ .lightning-cli decode tU-RLjMiDpY2U0o3W1oFowar36RFGpWloPbW9-
↪RuZdo9MyZpZD0wMjRiOWExZmE4ZTAwNmYxZTM5MzdmNjVmNjZjNDA4ZTZkYThlMWNhNzI4ZWE0MzIyMmE3MzgxZGYxY2M0NDk2N
```

(continues on next page)

(continued from previous page)

```
{
   "type": "rune",
   "unique_id": "3",
   "string": "b54f912e33220e9636534a375b5a05a306abdfa4451a95a5a0f6d6f7e46e65da:=3&
↪id=024b9a1fa8e006f1e3937f65f66c408e6da8e1ca728ea43222a7381df1cc449605&
↪method=listpeers&pnum=1&pnameid^024b9a1fa8e006f1e393|parr0^024b9a1fa8e006f1e393&time
↪<1656920538&rate=2",
   "restrictions": [
      {
         "alternatives": [
            "id=024b9a1fa8e006f1e3937f65f66c408e6da8e1ca728ea43222a7381df1cc449605"
         ],
         "summary": "id (of commanding peer) equal to
↪'024b9a1fa8e006f1e3937f65f66c408e6da8e1ca728ea43222a7381df1cc449605'"
      },
      {
         "alternatives": [
            "method=listpeers"
         ],
         "summary": "method (of command) equal to 'listpeers'"
      },
      {
         "alternatives": [
            "pnum=1"
         ],
         "summary": "pnum (number of command parameters) equal to 1"
      },
      {
         "alternatives": [
            "pnameid^024b9a1fa8e006f1e393",
            "parr0^024b9a1fa8e006f1e393"
         ],
         "summary": "pnameid (object parameter 'id') starts with '024b9a1fa8e006f1e393
↪' OR parr0 (array parameter #0) starts with '024b9a1fa8e006f1e393'"
      },
      {
         "alternatives": [
            "time<1656920538"
         ],
         "summary": "time (in seconds since 1970) less than 1656920538 (approximately␣
↪19 hours 18 minutes from now)"
      },
      {
         "alternatives": [
            "rate=2"
         ],
         "summary": "rate (max per minute) equal to 2"
      }
   ],
   "valid": true
}
```

24.5 SHARING RUNES

Because anyone can add a restriction to a rune, you can always turn a normal rune into a read-only rune, or restrict access for 30 minutes from the time you give it to someone. Adding restrictions before sharing runes is best practice.

If a rune has a ratelimit, any derived rune will have the same id, and thus will compete for that ratelimit. You might want to consider adding a tighter ratelimit to a rune before sharing it, so you will keep the remainder. For example, if you rune has a limit of 60 times per minute, adding a limit of 5 times per minute and handing that rune out means you can still use your original rune 55 times per minute.

24.6 RETURN VALUE

On success, an object is returned, containing:

- **rune** (string): the resulting rune
- **unique_id** (string): the id of this rune: this is set at creation and cannot be changed (even as restrictions are added)

The following warnings may also be returned:

- **warning_unrestricted_rune**: A warning shown when runes are created with powers that could drain your node

24.7 AUTHOR

Rusty Russell <rusty@rustcorp.com.au> wrote the original Python commando.py plugin, the in-tree commando plugin, and this manual page.

Christian Decker came up with the name "commando", which almost excuses his previous adoption of the name "Eltoo".

24.8 SEE ALSO

lightning-commando(7), lightning-decode(7)

24.9 RESOURCES

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

CHAPTER 25

lightning-connect – Command for connecting to another lightning node

25.1 SYNOPSIS

connect *id* [*host*] [*port*]

25.2 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* depends on the current network:

- bitcoin **mainnet**: 9735.
- bitcoin **testnet**: 19735.
- bitcoin **signet**: 39735.
- bitcoin **regtest**: 19846.

If *host* is not specified (or doesn't work), the connection will be attempted to an IP belonging to *id* obtained through gossip with other already connected peers. This can fail if your C-lightning node is a fresh install that has not connected to any peers yet (your node has no gossip yet), or if the target *id* is a fresh install that has no channels yet (nobody will gossip about a node until it has one published channel).

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).

If there are active channels with the peer, **connect** returns once all the subdaemons are in place to handle the channels, not just once it's connected.

25.3 RETURN VALUE

On success, an object is returned, containing:

- **id** (pubkey): the peer we connected to
- **features** (hex): BOLT 9 features bitmap offered by peer
- **direction** (string): Whether they initiated connection or we did (one of "in", "out")
- **address** (object): Address information (mainly useful if **direction** is *out*):
  - **type** (string): Type of connection (*torv2/torv3* only if **direction** is *out*) (one of "local socket", "ipv4", "ipv6", "torv2", "torv3")

  If **type** is "local socket":

  - **socket** (string): socket filename

  If **type** is "ipv4", "ipv6", "torv2" or "torv3":

  - **address** (string): address in expected format for **type**
  - **port** (u16): port number

25.4 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 peer disconnected while we were connecting:

```
{ "code" : 402, "message" : "..." }
```

If the given parameters are wrong:

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

25.5 AUTHOR

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

25.6 SEE ALSO

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

25.7 RESOURCES

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

CHAPTER 26

lightning-createinvoice – Low-level invoice creation

26.1 SYNOPSIS

createinvoice *invstring label preimage*

26.2 DESCRIPTION

The **createinvoice** RPC command signs and saves an invoice into the database.

The *invstring* parameter is of bolt11 form, but without the final signature appended. Minimal sanity checks are done. (Note: if **experimental-offers** is enabled, *invstring* can actually be an unsigned bolt12 invoice).

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 *preimage* is the preimage to supply upon successful payment of the invoice.

26.3 RETURN VALUE

(Note: the return format is the same as lightning-listinvoices(7)).

On success, an object is returned, containing:

- **label** (string): the label for the invoice
- **payment_hash** (hash): the hash of the *payment_preimage* which will prove payment (always 64 characters)
- **status** (string): Whether it has been paid, or can no longer be paid (one of "paid", "expired", "unpaid")
- **description** (string): Description extracted from **bolt11** or **bolt12**
- **expires_at** (u64): UNIX timestamp of when invoice expires (or expired)
- **bolt11** (string, optional): the bolt11 string (always present unless **bolt12** is)

- **bolt12** (string, optional): the bolt12 string instead of **bolt11** (**experimental-offers** only)
- **amount_msat** (msat, optional): The amount of the invoice (if it has one)
- **pay_index** (u64, optional): Incrementing id for when this was paid (**status** *paid* only)
- **amount_received_msat** (msat, optional): Amount actually received (**status** *paid* only)
- **paid_at** (u64, optional): UNIX timestamp of when invoice was paid (**status** *paid* only)
- **payment_preimage** (secret, optional): the proof of payment: SHA256 of this **payment_hash** (always 64 characters)
- **local_offer_id** (hex, optional): the *id* of our offer which created this invoice (**experimental-offers** only). (always 64 characters)
- **invreq_payer_note** (string, optional): the optional *invreq_payer_note* from invoice_request which created this invoice (**experimental-offers** only).

On failure, an error is returned and no invoice is created. If the lightning process fails before responding, the caller should use lightning-listinvoices(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.

26.4 AUTHOR

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

26.5 SEE ALSO

lightning-invoice(7), lightning-listinvoices(7), lightning-delinvoice(7), lightning-getroute(7), lightning-sendpay(7), lightning-offer(7).

26.6 RESOURCES

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

CHAPTER 27

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

27.1 SYNOPSIS

createonion *hops assocdata* [*session_key*] [*onion_size*]

27.2 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 Core 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": "11020203e904017b06080000670000010001"
        }, {
                "pubkey":
↪"035d2b1192dfba134e10e540875d366ebc8bc353d5aa766b80c090b39c3a5d885d",
                "payload": "11020203e804017506080000670000030001"
        }, {
                "pubkey":
↪"0382ce59ebf18be7d84677c2e35f23294b9992ceca95491fcf8a56c6cb2d9de199",
                "payload": "07020203e8040175"
        }
]
```

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,
        }, {
                "id":
↪"035d2b1192dfba134e10e540875d366ebc8bc353d5aa766b80c090b39c3a5d885d",
                "channel": "103x1x1",
                "direction": 0,
                "msatoshi": 1001,
                "amount_msat": "1001msat",
                "delay": 15,
        }, {
                "id":
↪"0382ce59ebf18be7d84677c2e35f23294b9992ceca95491fcf8a56c6cb2d9de199",
                "channel": "103x3x1",
                "direction": 0,
                "msatoshi": 1000,
                "amount_msat": "1000msat",
                "delay": 9,
        }
]
```

- Notice that the payload in the *hops* parameter is the hex-encoded TLV of the parameters in the `getroute` response, with length prepended as a `bigsize_t`.
- Except for the pubkey, the values are shifted left by one, i.e., the 1st payload in `createonion` corresponds to the 2nd set of values 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.

The optional *onion_size* parameter specifies a size different from the default payment onion (1300 bytes). May be used for custom protocols like trampoline routing.

27.3 RETURN VALUE

On success, an object is returned, containing:

- **onion** (hex): the onion packet (*onion_size* bytes)
- **shared_secrets** (array of secrets): one shared secret for each node in the *hops* parameter:
  - the shared secret with this hop (always 64 characters)

27.4 EXAMPLE

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**.

27.5 AUTHOR

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

27.6 SEE ALSO

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

27.7 RESOURCES

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

CHAPTER 28

lightning-datastore – Command for storing (plugin) data

28.1 SYNOPSIS

datastore *key* [*string*] [*hex*] [*mode*] [*generation*]

28.2 DESCRIPTION

The **datastore** RPC command allows plugins to store data in the Core Lightning database, for later retrieval.

key is an array of values (though a single value is treated as a one-element array), to form a hierarchy. Using the first element of the key as the plugin name (e.g. `[ "summary" ]`) is recommended. A key can either have children or a value, never both: parents are created and removed automatically.

mode is one of "must-create" (default, fails if it already exists), "must-replace" (fails if it doesn't already exist), "create-or-replace" (never fails), "must-append" (must already exist, append this to what's already there) or "create-or-append" (append if anything is there, otherwise create).

generation, if specified, means that the update will fail if the previously-existing data is not exactly that generation. This allows for simple atomicity. This is only legal with *mode* "must-replace" or "must-append".

28.3 RETURN VALUE

On success, an object is returned, containing:

- **key** (array of strings):
  - Part of the key added to the datastore
- **generation** (u64, optional): The number of times this has been updated
- **hex** (hex, optional): The hex data which has been added to the datastore
- **string** (string, optional): The data as a string, if it's valid utf-8

The following error codes may occur:

- 1202: The key already exists (and mode said it must not)
- 1203: The key does not exist (and mode said it must)
- 1204: The generation was wrong (and generation was specified)
- 1205: The key has children already.
- 1206: One of the parents already exists with a value.
- -32602: invalid parameters

28.4 AUTHOR

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

28.5 SEE ALSO

lightning-listdatastore(7), lightning-deldatastore(7)

28.6 RESOURCES

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

CHAPTER 29

lightning-decode – Command for decoding an invoice string (low-level)

29.1 SYNOPSIS

decode *string*

29.2 DESCRIPTION

The **decode** RPC command checks and parses:

- a *bolt11* or *bolt12* string (optionally prefixed by `lightning:` or `LIGHTNING:`) as specified by the BOLT 11 and BOLT 12 specifications.
- a *rune* as created by lightning-commando-rune(7).

It may decode other formats in future.

29.3 RETURN VALUE

On success, an object is returned, containing:

- **type** (string): what kind of object it decoded to (one of "bolt12 offer", "bolt12 invoice", "bolt12 invoice_request", "bolt11 invoice", "rune")
- **valid** (boolean): if this is false, you *MUST* not use the result except for diagnostics!

If **type** is "bolt12 offer", and **valid** is *true*:

- **offer_id** (hex): the id we use to identify this offer (always 64 characters)
- **offer_description** (string): the description of the purpose of the offer
- **offer_node_id** (pubkey): public key of the offering node

- **offer_chains** (array of hexs, optional): which blockchains this offer is for (missing implies bitcoin mainnet only):
  - the genesis blockhash (always 64 characters)
- **offer_metadata** (hex, optional): any metadata the creater of the offer includes
- **offer_currency** (string, optional): ISO 4217 code of the currency (missing implies Bitcoin) (always 3 characters)
- **currency_minor_unit** (u32, optional): the number of decimal places to apply to amount (if currency known)
- **offer_amount** (u64, optional): the amount in the `offer_currency` adjusted by `currency_minor_unit`, if any
- **offer_amount_msat** (msat, optional): the amount in bitcoin (if specified, and no `offer_currency`)
- **offer_issuer** (string, optional): the description of the creator of the offer
- **offer_features** (hex, optional): the feature bits of the offer
- **offer_absolute_expiry** (u64, optional): UNIX timestamp of when this offer expires
- **offer_quantity_max** (u64, optional): the maximum quantity (or, if 0, means any quantity)
- **offer_paths** (array of objects, optional): Paths to the destination:
  - **first_node_id** (pubkey): the (presumably well-known) public key of the start of the path
  - **blinding** (pubkey): blinding factor for this path
  - **path** (array of objects): an individual path:
    - **blinded_node_id** (pubkey): node_id of the hop
    - **encrypted_recipient_data** (hex): encrypted TLV entry for this hop
- **offer_recurrence** (object, optional): how often to this offer should be used:
  - **time_unit** (u32): the BOLT12 time unit
  - **period** (u32): how many `time_unit` per payment period
  - **time_unit_name** (string, optional): the name of `time_unit` (if valid)
  - **basetime** (u64, optional): period starts at this UNIX timestamp
  - **start_any_period** (u64, optional): you can start at any period (only if `basetime` present)
  - **limit** (u32, optional): maximum period number for recurrence
  - **paywindow** (object, optional): when within a period will payment be accepted (default is prior and during the period):
    - **seconds_before** (u32): seconds prior to period start
    - **seconds_after** (u32): seconds after to period start
    - **proportional_amount** (boolean, optional): amount should be scaled if payed after period start (always *true*)
- **unknown_offer_tlvs** (array of objects, optional): Any extra fields we didn't know how to parse:
  - **type** (u64): The type
  - **length** (u64): The length
  - **value** (hex): The value
- the following warnings are possible:

  - **warning_unknown_offer_currency**: The currency code is unknown (so no `currency_minor_unit`)

If **type** is "bolt12 offer", and **valid** is *false*:

- the following warnings are possible:
  - **warning_missing_offer_node_id**: `offer_node_id` is not present
  - **warning_invalid_offer_description**: `offer_description` is not valid UTF8
  - **warning_missing_offer_description**: `offer_description` is not present
  - **warning_invalid_offer_currency**: `offer_currency_code` is not valid UTF8
  - **warning_invalid_offer_issuer**: `offer_issuer` is not valid UTF8

If **type** is "bolt12 invoice_request", and **valid** is *true*:

- **offer_description** (string): the description of the purpose of the offer
- **offer_node_id** (pubkey): public key of the offering node
- **invreq_metadata** (hex): the payer-provided blob to derive invreq_payer_id
- **invreq_payer_id** (hex): the payer-provided key
- **signature** (bip340sig): BIP-340 signature of the `invreq_payer_id` on this invoice_request
- **offer_id** (hex, optional): the id we use to identify this offer (always 64 characters)
- **offer_chains** (array of hexs, optional): which blockchains this offer is for (missing implies bitcoin mainnet only):
  - the genesis blockhash (always 64 characters)
- **offer_metadata** (hex, optional): any metadata the creator of the offer includes
- **offer_currency** (string, optional): ISO 4217 code of the currency (missing implies Bitcoin) (always 3 characters)
- **currency_minor_unit** (u32, optional): the number of decimal places to apply to amount (if currency known)
- **offer_amount** (u64, optional): the amount in the `offer_currency` adjusted by `currency_minor_unit`, if any
- **offer_amount_msat** (msat, optional): the amount in bitcoin (if specified, and no `offer_currency`)
- **offer_issuer** (string, optional): the description of the creator of the offer
- **offer_features** (hex, optional): the feature bits of the offer
- **offer_absolute_expiry** (u64, optional): UNIX timestamp of when this offer expires
- **offer_quantity_max** (u64, optional): the maximum quantity (or, if 0, means any quantity)
- **offer_paths** (array of objects, optional): Paths to the destination:
  - **first_node_id** (pubkey): the (presumably well-known) public key of the start of the path
  - **blinding** (pubkey): blinding factor for this path
  - **path** (array of objects): an individual path:
    * **blinded_node_id** (pubkey): node_id of the hop
    * **encrypted_recipient_data** (hex): encrypted TLV entry for this hop
- **offer_recurrence** (object, optional): how often to this offer should be used:
  - **time_unit** (u32): the BOLT12 time unit

  - **period** (u32): how many `time_unit` per payment period
  - **time_unit_name** (string, optional): the name of `time_unit` (if valid)
  - **basetime** (u64, optional): period starts at this UNIX timestamp
  - **start_any_period** (u64, optional): you can start at any period (only if `basetime` present)
  - **limit** (u32, optional): maximum period number for recurrence
  - **paywindow** (object, optional): when within a period will payment be accepted (default is prior and during the period):
    * **seconds_before** (u32): seconds prior to period start
    * **seconds_after** (u32): seconds after to period start
    * **proportional_amount** (boolean, optional): amount should be scaled if payed after period start (always *true*)
- **invreq_chain** (hex, optional): which blockchain this offer is for (missing implies bitcoin mainnet only) (always 64 characters)
- **invreq_amount_msat** (msat, optional): the amount the invoice should be for
- **invreq_features** (hex, optional): the feature bits of the invoice_request
- **invreq_quantity** (u64, optional): the number of items to invoice for
- **invreq_payer_note** (string, optional): a note attached by the payer
- **invreq_recurrence_counter** (u32, optional): which number request this is for the same invoice
- **invreq_recurrence_start** (u32, optional): when we're requesting to start an invoice at a non-zero period
- **unknown_invoice_request_tlvs** (array of objects, optional): Any extra fields we didn't know how to parse:
  - **type** (u64): The type
  - **length** (u64): The length
  - **value** (hex): The value
- the following warnings are possible:
  - **warning_unknown_offer_currency**: The currency code is unknown (so no `currency_minor_unit`)

If **type** is "bolt12 invoice_request", and **valid** is *false*:

- the following warnings are possible:
  - **warning_invalid_offer_description**: `offer_description` is not valid UTF8
  - **warning_missing_offer_description**: `offer_description` is not present
  - **warning_invalid_offer_currency**: `offer_currency_code` is not valid UTF8
  - **warning_invalid_offer_issuer**: `offer_issuer` is not valid UTF8
  - **warning_missing_invreq_metadata**: `invreq_metadata` is not present
  - **warning_missing_invreq_payer_id**: `invreq_payer_id` is not present
  - **warning_invalid_invreq_payer_note**: `invreq_payer_note` is not valid UTF8
  - **warning_missing_invoice_request_signature**: `signature` is not present
  - **warning_invalid_invoice_request_signature**: Incorrect `signature`

If **type** is "bolt12 invoice", and **valid** is *true*:

- **offer_description** (string): the description of the purpose of the offer
- **offer_node_id** (pubkey): public key of the offering node
- **invreq_metadata** (hex): the payer-provided blob to derive invreq_payer_id
- **invreq_payer_id** (hex): the payer-provided key
- **invoice_paths** (array of objects): Paths to pay the destination:
  - **first_node_id** (pubkey): the (presumably well-known) public key of the start of the path
  - **blinding** (pubkey): blinding factor for this path
  - **path** (array of objects): an individual path:
    * **blinded_node_id** (pubkey): node_id of the hop
    * **encrypted_recipient_data** (hex): encrypted TLV entry for this hop
    * **fee_base_msat** (msat, optional): basefee for path
    * **fee_proportional_millionths** (u32, optional): proportional fee for path
    * **cltv_expiry_delta** (u32, optional): CLTV delta for path
    * **features** (hex, optional): features allowed for path
- **invoice_created_at** (u64): the UNIX timestamp of invoice creation
- **invoice_payment_hash** (hex): the hash of the *payment_preimage* (always 64 characters)
- **invoice_amount_msat** (msat): the amount required to fulfill invoice
- **signature** (bip340sig): BIP-340 signature of the `offer_node_id` on this invoice
- **offer_id** (hex, optional): the id we use to identify this offer (always 64 characters)
- **offer_chains** (array of hexs, optional): which blockchains this offer is for (missing implies bitcoin mainnet only):
  - the genesis blockhash (always 64 characters)
- **offer_metadata** (hex, optional): any metadata the creator of the offer includes
- **offer_currency** (string, optional): ISO 4217 code of the currency (missing implies Bitcoin) (always 3 characters)
- **currency_minor_unit** (u32, optional): the number of decimal places to apply to amount (if currency known)
- **offer_amount** (u64, optional): the amount in the `offer_currency` adjusted by `currency_minor_unit`, if any
- **offer_amount_msat** (msat, optional): the amount in bitcoin (if specified, and no `offer_currency`)
- **offer_issuer** (string, optional): the description of the creator of the offer
- **offer_features** (hex, optional): the feature bits of the offer
- **offer_absolute_expiry** (u64, optional): UNIX timestamp of when this offer expires
- **offer_quantity_max** (u64, optional): the maximum quantity (or, if 0, means any quantity)
- **offer_paths** (array of objects, optional): Paths to the destination:
  - **first_node_id** (pubkey): the (presumably well-known) public key of the start of the path
  - **blinding** (pubkey): blinding factor for this path
  - **path** (array of objects): an individual path:

    * **blinded_node_id** (pubkey): node_id of the hop
    * **encrypted_recipient_data** (hex): encrypted TLV entry for this hop

- **offer_recurrence** (object, optional): how often to this offer should be used:
  - **time_unit** (u32): the BOLT12 time unit
  - **period** (u32): how many `time_unit` per payment period
  - **time_unit_name** (string, optional): the name of `time_unit` (if valid)
  - **basetime** (u64, optional): period starts at this UNIX timestamp
  - **start_any_period** (u64, optional): you can start at any period (only if `basetime` present)
  - **limit** (u32, optional): maximum period number for recurrence
  - **paywindow** (object, optional): when within a period will payment be accepted (default is prior and during the period):
    * **seconds_before** (u32): seconds prior to period start
    * **seconds_after** (u32): seconds after to period start
    * **proportional_amount** (boolean, optional): amount should be scaled if payed after period start (always *true*)
- **invreq_chain** (hex, optional): which blockchain this offer is for (missing implies bitcoin mainnet only) (always 64 characters)
- **invreq_amount_msat** (msat, optional): the amount the invoice should be for
- **invreq_features** (hex, optional): the feature bits of the invoice_request
- **invreq_quantity** (u64, optional): the number of items to invoice for
- **invreq_payer_note** (string, optional): a note attached by the payer
- **invreq_recurrence_counter** (u32, optional): which number request this is for the same invoice
- **invreq_recurrence_start** (u32, optional): when we're requesting to start an invoice at a non-zero period
- **invoice_relative_expiry** (u32, optional): the number of seconds after *invoice_created_at* when this expires
- **invoice_fallbacks** (array of objects, optional): onchain addresses:
  - **version** (u8): Segwit address version
  - **hex** (hex): Raw encoded segwit address
  - **address** (string, optional): bech32 segwit address
- **invoice_features** (hex, optional): the feature bits of the invoice
- **invoice_node_id** (pubkey, optional): the id to pay (usually the same as offer_node_id)
- **invoice_recurrence_basetime** (u64, optional): the UNIX timestamp to base the invoice periods on
- **unknown_invoice_tlvs** (array of objects, optional): Any extra fields we didn't know how to parse:
  - **type** (u64): The type
  - **length** (u64): The length
  - **value** (hex): The value
- the following warnings are possible:
  - **warning_unknown_offer_currency**: The currency code is unknown (so no `currency_minor_unit`)

If **type** is "bolt12 invoice", and **valid** is *false*:

- **fallbacks** (array of objects, optional):
  - the following warnings are possible:
    - **warning_invoice_fallbacks_version_invalid**: `version` is > 16
- the following warnings are possible:
  - **warning_invalid_offer_description**: `offer_description` is not valid UTF8
  - **warning_missing_offer_description**: `offer_description` is not present
  - **warning_invalid_offer_currency**: `offer_currency_code` is not valid UTF8
  - **warning_invalid_offer_issuer**: `offer_issuer` is not valid UTF8
  - **warning_missing_invreq_metadata**: `invreq_metadata` is not present
  - **warning_invalid_invreq_payer_note**: `invreq_payer_note` is not valid UTF8
  - **warning_missing_invoice_paths**: `invoice_paths` is not present
  - **warning_missing_invoice_blindedpay**: `invoice_blindedpay` is not present
  - **warning_missing_invoice_created_at**: `invoice_created_at` is not present
  - **warning_missing_invoice_payment_hash**: `invoice_payment_hash` is not present
  - **warning_missing_invoice_amount**: `invoice_amount` is not present
  - **warning_missing_invoice_recurrence_basetime**: `invoice_recurrence_basetime` is not present
  - **warning_missing_invoice_node_id**: `invoice_node_id` is not present
  - **warning_missing_invoice_signature**: `signature` is not present
  - **warning_invalid_invoice_signature**: Incorrect `signature`

If **type** is "bolt11 invoice", and **valid** is *true*:

- **currency** (string): the BIP173 name for the currency
- **created_at** (u64): the UNIX-style timestamp of the invoice
- **expiry** (u64): the number of seconds this is valid after `created_at`
- **payee** (pubkey): the public key of the recipient
- **payment_hash** (hex): the hash of the *payment_preimage* (always 64 characters)
- **signature** (signature): signature of the *payee* on this invoice
- **min_final_cltv_expiry** (u32): the minimum CLTV delay for the final node
- **amount_msat** (msat, optional): Amount the invoice asked for
- **description** (string, optional): the description of the purpose of the purchase
- **description_hash** (hex, optional): the hash of the description, in place of *description* (always 64 characters)
- **payment_secret** (hex, optional): the secret to hand to the payee node (always 64 characters)
- **features** (hex, optional): the features bitmap for this invoice
- **payment_metadata** (hex, optional): the payment_metadata to put in the payment
- **fallbacks** (array of objects, optional): onchain addresses:

  - **type** (string): the address type (if known) (one of "P2PKH", "P2SH", "P2WPKH", "P2WSH")
  - **hex** (hex): Raw encoded address
  - **addr** (string, optional): the address in appropriate format for *type*
- **routes** (array of arrays, optional): Route hints to the *payee*:
  - hops in the route:
    * **pubkey** (pubkey): the public key of the node
    * **short_channel_id** (short_channel_id): a channel to the next peer
    * **fee_base_msat** (msat): the base fee for payments
    * **fee_proportional_millionths** (u32): the parts-per-million fee for payments
    * **cltv_expiry_delta** (u32): the CLTV delta across this hop
- **extra** (array of objects, optional): Any extra fields we didn't know how to parse:
  - **tag** (string): The bech32 letter which identifies this field (always 1 characters)
  - **data** (string): The bech32 data for this field

If **type** is "rune", and **valid** is *true*:

- **valid** (boolean) (always *true*)
- **string** (string): the string encoding of the rune
- **restrictions** (array of objects): restrictions built into the rune: all must pass:
  - **alternatives** (array of strings): each way restriction can be met: any can pass:
    * the alternative of form fieldname condition fieldname
  - **summary** (string): human-readable summary of this restriction
- **unique_id** (string, optional): unique id (always a numeric id on runes we create)
- **version** (string, optional): rune version, not currently set on runes we create

If **type** is "rune", and **valid** is *false*:

- **valid** (boolean) (always *false*)
- **hex** (hex, optional): the raw rune in hex
- the following warnings are possible:
  - **warning_rune_invalid_utf8**: the rune contains invalid UTF-8 strings

29.4 AUTHOR

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

29.5 SEE ALSO

lightning-pay(7), lightning-offer(7), lightning-offerout(7), lightning-fetchinvoice(7), lightning-sendinvoice(7), lightning-commando-rune(7)

BOLT #11.

BOLT #12.

29.6 RESOURCES

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

CHAPTER 30

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

30.1 SYNOPSIS

decodepay *bolt11* [*description*]

30.2 DESCRIPTION

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

30.3 RETURN VALUE

On success, an object is returned, containing:

- **currency** (string): the BIP173 name for the currency
- **created_at** (u64): the UNIX-style timestamp of the invoice
- **expiry** (u64): the number of seconds this is valid after *timestamp*
- **payee** (pubkey): the public key of the recipient
- **payment_hash** (hex): the hash of the *payment_preimage* (always 64 characters)
- **signature** (signature): signature of the *payee* on this invoice
- **min_final_cltv_expiry** (u32): the minimum CLTV delay for the final node
- **amount_msat** (msat, optional): Amount the invoice asked for
- **description** (string, optional): the description of the purpose of the purchase
- **description_hash** (hex, optional): the hash of the description, in place of *description* (always 64 characters)
- **payment_secret** (hex, optional): the secret to hand to the payee node (always 64 characters)

- **features** (hex, optional): the features bitmap for this invoice
- **payment_metadata** (hex, optional): the payment_metadata to put in the payment
- **fallbacks** (array of objects, optional): onchain addresses:
  - **type** (string): the address type (if known) (one of "P2PKH", "P2SH", "P2WPKH", "P2WSH")
  - **hex** (hex): Raw encoded address
  - **addr** (string, optional): the address in appropriate format for *type*
- **routes** (array of arrays, optional): Route hints to the *payee*:
  - hops in the route:
    - **pubkey** (pubkey): the public key of the node
    - **short_channel_id** (short_channel_id): a channel to the next peer
    - **fee_base_msat** (msat): the base fee for payments
    - **fee_proportional_millionths** (u32): the parts-per-million fee for payments
    - **cltv_expiry_delta** (u32): the CLTV delta across this hop
- **extra** (array of objects, optional): Any extra fields we didn't know how to parse:
  - **tag** (string): The bech32 letter which identifies this field (always 1 characters)
  - **data** (string): The bech32 data for this field

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.

30.4 AUTHOR

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

30.5 SEE ALSO

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

BOLT #11.

30.6 RESOURCES

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

CHAPTER 31

lightning-deldatastore – Command for removing (plugin) data

31.1 SYNOPSIS

deldatastore *key* [*generation*]

31.2 DESCRIPTION

The **deldatastore** RPC command allows plugins to delete data it has stored in the Core Lightning database.

The command fails if the *key* isn't present, or if *generation* is specified and the generation of the data does not exactly match.

31.3 RETURN VALUE

On success, an object is returned, containing:

- **key** (array of strings):
  - Part of the key added to the datastore
- **generation** (u64, optional): The number of times this has been updated
- **hex** (hex, optional): The hex data which has removed from the datastore
- **string** (string, optional): The data as a string, if it's valid utf-8

The following error codes may occur:

- 1200: the key does not exist
- 1201: the key does exist, but the generation is wrong
- -32602: invalid parameters

31.4 AUTHOR

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

31.5 SEE ALSO

lightning-listdatastore(7), lightning-datastore(7)

31.6 RESOURCES

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

CHAPTER 32

lightning-delexpiredinvoice – Command for removing expired invoices

32.1 SYNOPSIS

delexpiredinvoice [*maxexpirytime*]

32.2 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.

32.3 RETURN VALUE

On success, an empty object is returned.

32.4 AUTHOR

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

32.5 SEE ALSO

lightning-delinvoice(7), lightning-autoclean-status(7)

32.6 RESOURCES

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

CHAPTER 33

lightning-delforward – Command for removing a forwarding entry

33.1 SYNOPSIS

delforward *in_channel in_htlc_id status*

33.2 DESCRIPTION

The **delforward** RPC command removes a single forward from **listforwards**, using the uniquely-identifying *in_channel* and *in_htlc_id* (and, as a sanity check, the *status*) given by that command.

This command is mainly used by the *autoclean* plugin (see lightningd-config(7)), As these database entries are only kept for your own analysis, removing them has no effect on the running of your node.

You cannot delete forwards which have status *offered* (i.e. are currently active).

Note: for **listforwards** entries without an *in_htlc_id* entry (no longer created in v22.11, but can exist from older versions), a value of 18446744073709551615 can be used, but then it will delete *all* entries without *in_htlc_id* for this *in_channel* and *status*.

33.3 RETURN VALUE

On success, an empty object is returned.

33.4 ERRORS

The following errors may be reported:

- 1401: The forward specified does not exist.

33.5 AUTHOR

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

33.6 SEE ALSO

lightning-autoclean(7)

33.7 RESOURCES

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

CHAPTER 34

lightning-delinvoice – Command for removing an invoice (or just its description)

34.1 SYNOPSIS

delinvoice *label status* [*desconly*]

34.2 DESCRIPTION

The **delinvoice** RPC command removes an invoice with *status* as given in **listinvoices**, or with *desconly* set, removes its description.

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

If *desconly* is set, the invoice is not deleted, but has its description removed (this can save space with very large descriptions, as would be used with lightning-invoice(7) *deschashonly*.

34.3 RETURN VALUE

Note: The return is the same as an object from lightning-listinvoice(7).

On success, an object is returned, containing:

- **label** (string): Unique label given at creation time
- **payment_hash** (hash): the hash of the *payment_preimage* which will prove payment (always 64 characters)
- **status** (string): State of invoice (one of "paid", "expired", "unpaid")
- **expires_at** (u64): UNIX timestamp when invoice expires (or expired)
- **bolt11** (string, optional): BOLT11 string

- **bolt12** (string, optional): BOLT12 string
- **amount_msat** (msat, optional): the amount required to pay this invoice
- **description** (string, optional): description used in the invoice

If **bolt12** is present:

- **local_offer_id** (hex, optional): offer for which this invoice was created
- **invreq_payer_note** (string, optional): the optional *invreq_payer_note* from invoice_request which created this invoice

If **status** is "paid":

- **pay_index** (u64): unique index for this invoice payment
- **amount_received_msat** (msat): how much was actually received
- **paid_at** (u64): UNIX timestamp of when payment was received
- **payment_preimage** (secret): SHA256 of this is the *payment_hash* offered in the invoice (always 64 characters)

34.4 ERRORS

The following errors may be reported:

- -1: Database error.
- 905: An invoice with that label does not exist.
- 906: The invoice *status* does not match the parameter. An error object will be returned as error *data*, containing *current_status* and *expected_status* fields. This is most likely due to the *status* of the invoice changing just before this command is invoked.
- 908: The invoice already has no description, and *desconly* was set.

34.5 AUTHOR

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

34.6 SEE ALSO

lightning-listinvoice(7), lightning-waitinvoice(7), lightning-invoice(7), lightning-delexpiredinvoice(7), lightning-autoclean-status(7)

34.7 RESOURCES

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

CHAPTER 35

lightning-delpay – Command for removing a completed or failed payment

35.1 SYNOPSIS

delpay *payment_hash status* [*partid groupid*]

35.2 DESCRIPTION

The **delpay** RPC command deletes a payment with the given `payment_hash` if its status is either `complete` or `failed`. Deleting a `pending` payment is an error. If *partid* and *groupid* are not specified, all payment parts are deleted.

- *payment_hash*: The unique identifier of a payment.
- *status*: Expected status of the payment.
- *partid*: Specific partid to delete (must be paired with *groupid*)
- *groupid*: Specific groupid to delete (must be paired with *partid*)

Only deletes if the payment status matches.

35.3 EXAMPLE JSON REQUEST

```
{
  "id": 82,
  "method": "delpay",
  "params": {
    "payment_hash": "4fa2f1b001067ec06d7f95b8695b8acd9ef04c1b4d1110e3b94e1fa0687bb1e0
↪",
    "status": "complete"
  }
}
```

35.4 RETURN VALUE

The returned format is the same as lightning-listsendpays(7). If the payment is a multi-part payment (MPP) the command return a list of payments will be returned – one payment object for each partid.

On success, an object containing **payments** is returned. It is an array of objects, where each object contains:

- **id** (u64): unique ID for this payment attempt
- **payment_hash** (hex): the hash of the *payment_preimage* which will prove payment (always 64 characters)
- **status** (string): status of the payment (one of "pending", "failed", "complete")
- **amount_sent_msat** (msat): the amount we actually sent, including fees
- **created_at** (u64): the UNIX timestamp showing when this payment was initiated
- **partid** (u64, optional): unique ID within this (multi-part) payment
- **destination** (pubkey, optional): the final destination of the payment if known
- **amount_msat** (msat, optional): the amount the destination received, if known
- **completed_at** (u64, optional): the UNIX timestamp showing when this payment was completed
- **groupid** (u64, optional): Grouping key to disambiguate multiple attempts to pay an invoice or the same payment_hash
- **payment_preimage** (hex, optional): proof of payment (always 64 characters)
- **label** (string, optional): the label, if given to sendpay
- **bolt11** (string, optional): the bolt11 string (if pay supplied one)
- **bolt12** (string, optional): the bolt12 string (if supplied for pay: **experimental-offers** only).
- **erroronion** (hex, optional): the error onion returned on failure, if any.

On failure, an error is returned. If the lightning process fails before responding, the caller should use lightning-listsentpays(7) or lightning-listpays(7) to query whether this payment was deleted or not.

The following error codes may occur:

- -32602: Parameter missed or malformed;
- 211: Payment status mismatch. Check the correct status via **paystatus**;
- 208: Payment with payment_hash not found.

35.5 EXAMPLE JSON RESPONSE

```
{
   "payments": [
      {
         "id": 2,
         "payment_hash":
↪"8dfd6538eeb33811c9114a75f792a143728d7f05643f38c3d574d3097e8910c0",
         "destination":
↪"0219f8900ee78a89f050c24d8b69492954f9fdbabed753710845eb75d3a75a5880",
         "msatoshi": 1000,
         "amount_msat": "1000msat",
         "msatoshi_sent": 1000,
```

(continues on next page)

(continued from previous page)

```
         "amount_sent_msat": "1000msat",
         "created_at": 1596224858,
         "status": "complete",
         "payment_preimage":
↪"35bd4e2b481a1a84a22215b5372672cf81460a671816960ddb206464359e1822",
         "bolt11":
↪"lntb10n1p0jga20pp53h7k2w8wkvuprjg3ff6l0y4pgdeg6lc9vsln3s74wnfsjl5fzrqqdqdw3jhxazldahx2xqyjw5qcqp2
↪"
      }
   ]
}
```

35.6 AUTHOR

Vincenzo Palazzo <vincenzo.palazzo@protonmail.com> is mainly responsible.

35.7 SEE ALSO

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

35.8 RESOURCES

Main web site: https://github.com/ElementsProject/lightning *comment*: # (SHA256STAMP:1ce2241eeae759ed5566342fb7810e62fa2c618f2465314f17376ebe9b6d24f8)

CHAPTER 36

lightning-disableoffer – Command for removing an offer

36.1 SYNOPSIS

(WARNING: experimental-offers only)

disableoffer *offer_id*

36.2 DESCRIPTION

The **disableoffer** RPC command disables an offer, so that no further invoices will be given out (if made with lightning-offer(7)) or invoices accepted (if made with lightning-offerout(7)).

We currently don't support deletion of offers, so offers are not forgotten entirely (there may be invoices which refer to this offer).

36.3 EXAMPLE JSON REQUEST

```
{
  "id": 82,
  "method": "disableoffer",
  "params": {
    "offer_id": "713a16ccd4eb10438bdcfbc2c8276be301020dd9d489c530773ba64f3b33307d ",
  }
}
```

36.4 RETURN VALUE

Note: the returned object is the same format as **listoffers**.

On success, an object is returned, containing:

- **offer_id** (hex): the merkle hash of the offer (always 64 characters)
- **active** (boolean): Whether the offer can produce invoices/payments (always *false*)
- **single_use** (boolean): Whether the offer is disabled after first successful use
- **bolt12** (string): The bolt12 string representing this offer
- **used** (boolean): Whether the offer has had an invoice paid / payment made
- **label** (string, optional): The label provided when offer was created

36.5 EXAMPLE JSON RESPONSE

```
{
   "offer_id": "053a5c566fbea2681a5ff9c05a913da23e45b95d09ef5bd25d7d408f23da7084",
   "active": false,
   "single_use": false,
   "bolt12":
↪"lno1qgsqvgnwgcg35z6ee2h3yczraddm72xrfua9uve2rlrm9deu7xyfzrcgqvqcdgq2z9pk7enxv4jjqen0wgs8yatnw3ujz8
↪",
   "used": false
}
```

36.6 AUTHOR

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

36.7 SEE ALSO

lightning-offer(7), lightning-offerout(7), lightning-listoffers(7).

36.8 RESOURCES

Main web site: https://github.com/ElementsProject/lightning *comment*: # (SHA256STAMP:b471374a7c160373b328c2171953225b7fa27d26314a270e95320c1b6ef57307)

CHAPTER 37

lightning-disconnect – Command for disconnecting from another lightning node

37.1 SYNOPSIS

disconnect *id* [*force*]

37.2 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.

37.3 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.

37.4 AUTHOR

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

37.5 SEE ALSO

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

37.6 RESOURCES

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

CHAPTER 38

lightning-emergencyrecover – Command for recovering channels from the emergency.recovery file in the lightning directory

38.1 SYNOPSIS

emergencyrecover

38.2 DESCRIPTION

The **emergencyrecover** RPC command fetches data from the emergency.recover file and tries to reconnect to the peer and force him to close the channel. The data in this file has enough information to reconnect and sweep the funds.

This recovery method is not spontaneous and it depends on the peer, so it should be used as a last resort to recover the funds stored in a channel in case of severe data loss.

38.3 RETURN VALUE

On success, an object is returned, containing:

- **stubs** (array of hexs):
  - Each item is the channel ID of the channel successfully inserted

38.4 AUTHOR

Aditya <aditya.sharma20111@gmail.com> is mainly responsible.

38.5 SEE ALSO

lightning-getsharedsecret(7)

38.6 RESOURCES

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

CHAPTER 39

lightning-feerates – Command for querying recommended onchain feerates

39.1 SYNOPSIS

feerates *style*

39.2 DESCRIPTION

The **feerates** command returns the feerates that CLN will use. The feerates will be based on the recommended feerates from the backend. The backend may fail to provide estimates, but if it was able to provide estimates in the past, CLN will continue to use those for a while. CLN will also smoothen feerate estimations from the backend.

style is either of the two strings:

- *perkw* - provide feerate in units of satoshis per 1000 weight.
- *perkb* - provide feerate in units of satoshis per 1000 virtual bytes.

Bitcoin transactions have non-witness and witness bytes:

- Non-witness bytes count as 4 weight, 1 virtual byte. All bytes other than SegWit witness count as non-witness bytes.
- Witness bytes count as 1 weight, 0.25 virtual bytes.

Thus, all *perkb* feerates will be exactly 4 times *perkw* feerates.

To compute the fee for a transaction, multiply its weight or virtual bytes by the appropriate *perkw* or *perkw* feerate returned by this command, then divide by 1000.

There is currently no way to change these feerates from the RPC. If you need custom control over onchain feerates, you will need to provide your own plugin that replaces the `bcli` plugin backend. For commands like lightning-withdraw(7) or lightning-fundchannel(7) you can provide a preferred feerate directly as a parameter, which will override the recommended feerates returned by **feerates**.

39.3 RETURN VALUE

On success, an object is returned, containing:

- **perkb** (object, optional): If *style* parameter was perkb:
  - **min_acceptable** (u32): The smallest feerate that you can use, usually the minimum relayed feerate of the backend
  - **max_acceptable** (u32): The largest feerate we will accept from remote negotiations. If a peer attempts to set the feerate higher than this we will unilaterally close the channel (or simply forget it if it's not open yet).
  - **opening** (u32, optional): Default feerate for lightning-fundchannel(7) and lightning-withdraw(7)
  - **mutual_close** (u32, optional): Feerate to aim for in cooperative shutdown. Note that since mutual close is a **negotiation**, the actual feerate used in mutual close will be somewhere between this and the corresponding mutual close feerate of the peer.
  - **unilateral_close** (u32, optional): Feerate for commitment_transaction in a live channel which we originally funded
  - **delayed_to_us** (u32, optional): Feerate for returning unilateral close funds to our wallet
  - **htlc_resolution** (u32, optional): Feerate for returning unilateral close HTLC outputs to our wallet
  - **penalty** (u32, optional): Feerate to start at when penalizing a cheat attempt
- **perkw** (object, optional): If *style* parameter was perkw:
  - **min_acceptable** (u32): The smallest feerate that you can use, usually the minimum relayed feerate of the backend
  - **max_acceptable** (u32): The largest feerate we will accept from remote negotiations. If a peer attempts to set the feerate higher than this we will unilaterally close the channel (or simply forget it if it's not open yet).
  - **opening** (u32, optional): Default feerate for lightning-fundchannel(7) and lightning-withdraw(7)
  - **mutual_close** (u32, optional): Feerate to aim for in cooperative shutdown. Note that since mutual close is a **negotiation**, the actual feerate used in mutual close will be somewhere between this and the corresponding mutual close feerate of the peer.
  - **unilateral_close** (u32, optional): Feerate for commitment_transaction in a live channel which we originally funded
  - **delayed_to_us** (u32, optional): Feerate for returning unilateral close funds to our wallet
  - **htlc_resolution** (u32, optional): Feerate for returning unilateral close HTLC outputs to our wallet
  - **penalty** (u32, optional): Feerate to start at when penalizing a cheat attempt
- **onchain_fee_estimates** (object, optional):
  - **opening_channel_satoshis** (u64): Estimated cost of typical channel open
  - **mutual_close_satoshis** (u64): Estimated cost of typical channel close
  - **unilateral_close_satoshis** (u64): Estimated cost of typical unilateral close (without HTLCs)
  - **htlc_timeout_satoshis** (u64): Estimated cost of typical HTLC timeout transaction
  - **htlc_success_satoshis** (u64): Estimated cost of typical HTLC fulfillment transaction

The following warnings may also be returned:

- **warning_missing_feerates**: Some fee estimates are missing

39.4 ERRORS

The **feerates** command will never error, however some fields may be missing in the result if feerate estimates for that kind of transaction are unavailable.

39.5 NOTES

Many other commands have a *feerate* parameter, which can be the strings *urgent*, *normal*, or *slow*. These are mapped to the **feerates** outputs as:

- *urgent* - equal to *unilateral_close*
- *normal* - equal to *opening*
- *slow* - equal to *min_acceptable*.

39.6 TRIVIA

In C-lightning we like to call the weight unit "sipa" in honor of Pieter Wuille, who uses the name "sipa" on IRC and elsewhere. Internally we call the *perkw* style as "feerate per kilosipa".

39.7 AUTHOR

ZmnSCPxj <ZmnSCPxj@protonmail.com> wrote the initial version of this manpage.

39.8 SEE ALSO

lightning-parsefeerate(7), lightning-fundchannel(7), lightning-withdraw(7), lightning-txprepare(7), lightning-fundchannel_start(7).

39.9 RESOURCES

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

CHAPTER 40

lightning-fetchinvoice – Command for fetch an invoice for an offer

40.1 SYNOPSIS

(WARNING: experimental-offers only)

fetchinvoice *offer* [*msatoshi*] [*quantity*] [*recurrence_counter*] [*recurrence_start*] [*recurrence_label*] [*timeout*] [*payer_note*]

40.2 DESCRIPTION

The **fetchinvoice** RPC command contacts the issuer of an *offer* to get an actual invoice that can be paid. It highlights any changes between the offer and the returned invoice.

If **fetchinvoice-noconnect** is not specified in the configuation, it will connect to the destination in the (currently common!) case where it cannot find a route which supports `option_onion_messages`.

The offer must not contain *send_invoice*; see lightning-sendinvoice(7).

msatoshi is required if the *offer* does not specify an amount at all, otherwise it is not allowed.

quantity is is required if the *offer* specifies *quantity_min* or *quantity_max*, otherwise it is not allowed.

recurrence_counter is required if the *offer* specifies *recurrence*, otherwise it is not allowed. *recurrence_counter* should first be set to 0, and incremented for each successive invoice in a given series.

recurrence_start is required if the *offer* specifies *recurrence_base* with *start_any_period* set, otherwise it is not allowed. It indicates what period number to start at.

recurrence_label is required if *recurrence_counter* is set, and otherwise is not allowed. It must be the same as prior fetchinvoice calls for the same recurrence, as it is used to link them together.

timeout is an optional timeout; if we don't get a reply before this we fail (default, 60 seconds).

payer_note is an optional payer note to include in the fetched invoice.

40.3 RETURN VALUE

On success, an object is returned, containing:

- **invoice** (string): The BOLT12 invoice we fetched
- **changes** (object): Summary of changes from offer:
  - **description_appended** (string, optional): extra characters appended to the *description* field.
  - **description** (string, optional): a completely replaced *description* field
  - **vendor_removed** (string, optional): The *vendor* from the offer, which is missing in the invoice
  - **vendor** (string, optional): a completely replaced *vendor* field
  - **amount_msat** (msat, optional): the amount, if different from the offer amount multiplied by any *quantity* (or the offer had no amount, or was not in BTC).
- **next_period** (object, optional): Only for recurring invoices if the next period is under the *recurrence_limit*:
  - **counter** (u64): the index of the next period to fetchinvoice
  - **starttime** (u64): UNIX timestamp that the next period starts
  - **endtime** (u64): UNIX timestamp that the next period ends
  - **paywindow_start** (u64): UNIX timestamp of the earliest time that the next invoice can be fetched
  - **paywindow_end** (u64): UNIX timestamp of the latest time that the next invoice can be fetched

The following error codes may occur:

- -1: Catchall nonspecific error.
- 1002: Offer has expired.
- 1003: Cannot find a route to the node making the offer.
- 1004: The node making the offer returned an error message.
- 1005: We timed out trying to fetch an invoice.

40.4 AUTHOR

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

40.5 SEE ALSO

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

40.6 RESOURCES

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

CHAPTER 41

lightning-fundchannel – Command for establishing a lightning channel

41.1 SYNOPSIS

fundchannel *id* *amount* [*feerate*] [*announce*] [*minconf*] [*utxos*] [*push_msat*] [*close_to*] [*request_amt*] [*compact_lease*]

41.2 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. If not already connected, **fundchannel** will automatically attempt to connect if C-lightning knows a way to contact the node (either from normal gossip, or from a previous **connect** call). This auto-connection can fail if C-lightning does not know how to contact the target node; see lightning-connect(7). 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 and large channels were not negotiated with the peer). 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 (unless large channels were negotiated with the peer).

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.

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.

request_amt is an amount of liquidity you'd like to lease from the peer. If peer supports `option_will_fund`, indicates to them to include this much liquidity into the channel. Must also pass in *compact_lease*.

compact_lease is a compact represenation of the peer's expected channel lease terms. If the peer's terms don't match this set, we will fail to open the channel.

This example shows how to use lightning-cli to open new channel with peer 03f…fc1 from one whole utxo bcc1…39c:0 (you can use **listfunds** command to get txid and vout):

```
lightning-cli -k fundchannel id=03f...fc1 amount=all feerate=normal utxos='["bcc1...
↪39c:0"]'
```

41.3 RETURN VALUE

On success, an object is returned, containing:

- **tx** (hex): The raw transaction which funded the channel
- **txid** (txid): The txid of the transaction which funded the channel
- **outnum** (u32): The 0-based output index showing which output funded the channel
- **channel_id** (hex): The channel_id of the resulting channel (always 64 characters)
- **close_to** (hex, optional): The raw scriptPubkey which mutual close will go to; only present if *close_to* parameter was specified and peer supports `option_upfront_shutdown_script`
- **mindepth** (u32, optional): Number of confirmations before we consider the channel active.

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.).

41.4 SEE ALSO

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

41.5 RESOURCES

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

CHAPTER 42

lightning-fundchannel_cancel – Command for completing channel establishment

42.1 SYNOPSIS

fundchannel_cancel *id*

42.2 DESCRIPTION

`fundchannel_cancel` is a lower level RPC command. It allows channel opener 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.

42.3 RETURN VALUE

On success, an object is returned, containing:

- **cancelled** (string): A message indicating it was cancelled by RPC

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.
- 306: Unknown peer id.
- 307: No channel currently being funded that can be cancelled.

- 308: It is unsafe to cancel the channel: the funding transaction has been broadcast, or there are HTLCs already in the channel, or the peer was the initiator and not us.

42.4 AUTHOR

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

42.5 SEE ALSO

lightning-connect(7), lightning-fundchannel(7), lightning-multifundchannel(7), lightning-fundchannel_start(7), lightning-fundchannel_complete(7) lightning-openchannel_init(7), lightning-openchannel_update(7), lightning-openchannel_signed(7), lightning-openchannel_abort(7)

42.6 RESOURCES

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

CHAPTER 43

lightning-fundchannel_complete – Command for completing channel establishment

43.1 SYNOPSIS

fundchannel_complete *id psbt*

43.2 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.

psbt is the transaction to use for funding (does not need to be signed but must be otherwise complete).

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.

43.3 RETURN VALUE

On success, an object is returned, containing:

- **channel_id** (hex): The channel_id of the resulting channel (always 64 characters)
- **commitments_secured** (boolean): Indication that channel is safe to use (always *true*)

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.
- 309: PSBT does not have a unique, correct output to fund the channel.

43.4 AUTHOR

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

43.5 SEE ALSO

lightning-connect(7), lightning-fundchannel(7), lightning-multifundchannel(7), lightning-fundchannel_start(7), lightning-fundchannel_cancel(7), lightning-openchannel_init(7), lightning-openchannel_update(7), lightning-openchannel_signed(7), lightning-openchannel_bump(7), lightning-openchannel_abort(7)

43.6 RESOURCES

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

CHAPTER 44

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

44.1 SYNOPSIS

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

44.2 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.

44.3 RETURN VALUE

On success, an object is returned, containing:

- **funding_address** (string): The address to send funding to for the channel. DO NOT SEND COINS TO THIS ADDRESS YET.
- **scriptpubkey** (hex): The raw scriptPubkey for the address
- **close_to** (hex, optional): The raw scriptPubkey which mutual close will go to; only present if *close_to* parameter was specified and peer supports `option_upfront_shutdown_script`
- **mindepth** (u32, optional): Number of confirmations before we consider the channel active.

The following warnings may also be returned:

- **warning_usage**: A warning not to prematurely broadcast the funding transaction (always present!)

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.

44.4 AUTHOR

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

44.5 SEE ALSO

lightning-connect(7), lightning-fundchannel(7), lightning-multifundchannel(7), lightning-fundchannel_complete(7), lightning-fundchannel_cancel(7) lightning-openchannel_init(7), lightning-openchannel_update(7), lightning-openchannel_signed(7), lightning-openchannel_bump(7), lightning-openchannel_abort(7)

44.6 RESOURCES

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

CHAPTER 45

lightning-funderupdate – Command for adjusting node funding v2 channels

45.1 SYNOPSIS

funderupdate [*policy*] [*policy_mod*] [*leases_only*] [*min_their_funding_msat*] [*max_their_funding_msat*] [*per_channel_min_msat*] [*per_channel_max_msat*] [*reserve_tank_msat*] [*fuzz_percent*] [*fund_probability*] [*lease_fee_base_msat*] [*lease_fee_basis*] [*funding_weight*] [*channel_fee_max_base_msat*] [*channel_fee_max_proportional_thousandths*] [*compact_lease*]

NOTE: Must have –experimental-dual-fund enabled for these settings to take effect.

45.2 DESCRIPTION

For channel open requests using

policy, *policy_mod* is the policy the funder plugin will use to decide how much capital to commit to a v2 open channel request. There are three policy options, detailed below: `match`, `available`, and `fixed`. The *policy_mod* is the number or 'modification' to apply to the policy. Default is (fixed, 0sats).

- `match` – Contribute *policy_mod* percent of their requested funds. Valid *policy_mod* values are 0 to 200. If this is a channel lease request, we match based on their requested funds. If it is not a channel lease request (and *lease_only* is false), then we match their funding amount. Note: any lease match less than 100 will likely fail, as clients will not accept a lease less than their request.
- `available` – Contribute *policy_mod* percent of our available node wallet funds. Valid *policy_mod* values are 0 to 100.
- `fixed` – Contributes a fixed *policy_mod* sats to v2 channel open requests.

Note: to maximize channel leases, best policy setting is (match, 100).

leases_only will only contribute funds to `option_will_fund` requests which pay to lease funds. Defaults to false, will fund any v2 open request using *policy* even if it's they're not seeking to lease funds. Note that `option_will_fund` commits funds for 4032 blocks (~1mo). Must also set *lease_fee_base_msat*, *lease_fee_basis*, *funding_weight*, *channel_fee_max_base_msat*, and *channel_fee_max_proportional_thousandths* to advertise available channel leases.

min_their_funding_msat is the minimum funding sats that we require in order to activate our contribution policy to the v2 open. Defaults to 10k sats.

max_their_funding_msat is the maximum funding sats that we will consider to activate our contribution policy to the v2 open. Any channel open above this will not be funded. Defaults to no max (`UINT_MAX`).

per_channel_min_msat is the minimum amount that we will contribute to a channel open. Defaults to 10k sats.

per_channel_max_msat is the maximum amount that we will contribute to a channel open. Defaults to no max (`UINT_MAX`).

reserve_tank_msat is the amount of sats to leave available in the node wallet. Defaults to zero sats.

fuzz_percent is a percentage to fuzz the resulting contribution amount by. Valid values are 0 to 100. Note that turning this on with (match, 100) policy will randomly fail `option_will_fund` leases, as most clients expect an exact or greater match of their `requested_funds`. Defaults to 0% (no fuzz).

fund_probability is the percent of v2 channel open requests to apply our policy to. Valid values are integers from 0 (fund 0% of all open requests) to 100 (fund every request). Useful for randomizing opens that receive funds. Defaults to 100.

Setting any of the next 5 options will activate channel leases for this node, and advertise these values via the lightning gossip network. If any one is set, the other values will be the default.

lease_fee_base_msat is the flat fee for a channel lease. Node will receive this much extra added to their channel balance, paid by the opening node. Defaults to 2k sats. Note that the minimum is 1sat.

lease_fee_basis is a basis fee that's calculated as 1/10k of the total requested funds the peer is asking for. Node will receive the total of *lease_fee_basis* times requested funds / 10k satoshis added to their channel balance, paid by the opening node. Default is 0.65% (65 basis points)

funding_weight is used to calculate the fee the peer will compensate your node for its contributing inputs to the funding transaction. The total fee is calculated as the `open_channel2.funding_feerate_perkw` times this *funding_weight* divided by 1000. Node will have this funding fee added to their channel balance, paid by the opening node. Default is 2 inputs + 1 P2WPKH output.

channel_fee_max_base_msat is a commitment to a maximum `channel_fee_base_msat` that your node will charge for routing payments over this leased channel during the lease duration. Default is 5k sats.

channel_fee_max_proportional_thousandths is a commitment to a maximum `channel_fee_proportional_millionths` that your node will charge for routing payments over this leased channel during the lease duration. Note that it's denominated in 'thousandths'. A setting of `1` is equal to 1k ppm; `5` is 5k ppm, etc. Default is 100 (100k ppm).

compact_lease is a compact description of the channel lease params. When opening a channel, passed in to `fundchannel` to indicate the terms we expect from the peer.

45.3 RETURN VALUE

On success, an object is returned, containing:

- **summary** (string): Summary of the current funding policy e.g. (match 100)
- **policy** (string): Policy funder plugin will use to decide how much captial to commit to a v2 open channel request (one of "match", "available", "fixed")
- **policy_mod** (u32): The *policy_mod* is the number or 'modification' to apply to the policy.

- **leases_only** (boolean): Only contribute funds to `option_will_fund` lease requests.
- **min_their_funding_msat** (msat): The minimum funding sats that we require from peer to activate our funding policy.
- **max_their_funding_msat** (msat): The maximum funding sats that we'll allow from peer to activate our funding policy.
- **per_channel_min_msat** (msat): The minimum amount that we will fund a channel open with.
- **per_channel_max_msat** (msat): The maximum amount that we will fund a channel open with.
- **reserve_tank_msat** (msat): Amount of sats to leave available in the node wallet.
- **fuzz_percent** (u32): Percentage to fuzz our funding amount by.
- **fund_probability** (u32): Percent of opens to consider funding. 100 means we'll consider funding every requested open channel request.
- **lease_fee_base_msat** (msat, optional): Flat fee to charge for a channel lease.
- **lease_fee_basis** (u32, optional): Proportional fee to charge for a channel lease, calculated as 1/10,000th of requested funds.
- **funding_weight** (u32, optional): Transaction weight the channel opener will pay us for a leased funding transaction.
- **channel_fee_max_base_msat** (msat, optional): Maximum channel_fee_base_msat we'll charge for routing funds leased on this channel.
- **channel_fee_max_proportional_thousandths** (u32, optional): Maximum channel_fee_proportional_millitionths we'll charge for routing funds leased on this channel, in thousandths.
- **compact_lease** (hex, optional): Compact description of the channel lease parameters.

The following error code may occur:

- -32602: If the given parameters are invalid.

45.4 AUTHOR

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

45.5 SEE ALSO

lightning-fundchannel(7), lightning-listfunds(7)

45.6 RESOURCES

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

CHAPTER 46

lightning-fundpsbt – Command to populate PSBT inputs from the wallet

46.1 SYNOPSIS

fundpsbt *satoshi feerate startweight* [*minconf*] [*reserve*] [*locktime*] [*min_witness_weight*] [*excess_as_change*]

46.2 DESCRIPTION

`fundpsbt` is a low-level RPC command which creates a PSBT using unreserved inputs in the wallet, optionally reserving them as well.

satoshi is the minimum satoshi value of the output(s) needed (or the string "all" meaning use all unreserved inputs). If a value, 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 can be one of the feerates listed in lightning-feerates(7), or 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. It can also be a *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*.

startweight is the weight of the transaction before *fundpsbt* has added any inputs.

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

If *reserve* if not zero, then *reserveinputs* is called (successfully, with *exclusive* true) on the returned PSBT for this number of blocks (default 72 blocks if unspecified).

locktime is an optional locktime: if not set, it is set to a recent block height.

min_witness_weight is an optional minimum weight to use for a UTXO's witness. If the actual witness weight is greater than the provided minimum, the actual witness weight will be used.

excess_as_change is an optional boolean to flag to add a change output for the excess sats.

46.3 EXAMPLE USAGE

Let's assume the caller is trying to produce a 100,000 satoshi output.

First, the caller estimates the weight of the core (typically 42) and known outputs of the transaction (typically (9 + scriptlen) * 4). For a simple P2WPKH it's a 22 byte scriptpubkey, so that's 124 weight.

It calls "*fundpsbt* 100000sat slow 166", which succeeds, and returns the *psbt* and *feerate_per_kw* it used, the *estimated_final_weight* and any *excess_msat*.

If *excess_msat* is greater than the cost of adding a change output, the caller adds a change output randomly to position 0 or 1 in the PSBT. Say *feerate_per_kw* is 253, and the change output is a P2WPKH (weight 124), the cost is around 31 sats. With the dust limit disallowing payments below 546 satoshis, we would only create a change output if *excess_msat* was greater or equal to 31 + 546.

46.4 RETURN VALUE

On success, an object is returned, containing:

- **psbt** (string): Unsigned PSBT which fulfills the parameters given
- **feerate_per_kw** (u32): The feerate used to create the PSBT, in satoshis-per-kiloweight
- **estimated_final_weight** (u32): The estimated weight of the transaction once fully signed
- **excess_msat** (msat): The amount above *satoshi* which is available. This could be zero, or dust; it will be zero if *change_outnum* is also returned
- **change_outnum** (u32, optional): The 0-based output number where change was placed (only if parameter *excess_as_change* was true and there was sufficient funds)
- **reservations** (array of objects, optional): If *reserve* was true or a non-zero number, just as per lightning-reserveinputs(7):
  - **txid** (txid): The txid of the transaction
  - **vout** (u32): The 0-based output number
  - **was_reserved** (boolean): Whether this output was previously reserved (always *false*)
  - **reserved** (boolean): Whether this output is now reserved (always *true*)
  - **reserved_to_block** (u32): The blockheight the reservation will expire

If *excess_as_change* is true and the excess is enough to cover an additional output above the `dust_limit`, then an output is added to the PSBT for the excess amount. The *excess_msat* will be zero. A *change_outnum* will be returned with the index of the change output.

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.
- 301: Insufficient UTXOs to meet *satoshi* value.

46.5 AUTHOR

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

46.6 SEE ALSO

lightning-utxopsbt(7), lightning-reserveinputs(7), lightning-unreserveinputs(7).

46.7 RESOURCES

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

CHAPTER 47

lightning-getinfo – Command to receive all information about the Core Lightning node.

47.1 SYNOPSIS

getinfo

47.2 DESCRIPTION

The **getinfo** gives a summary of the current running node.

47.3 EXAMPLE JSON REQUEST

```
{
  "id": 82,
  "method": "getinfo",
  "params": {}
}
```

47.4 RETURN VALUE

On success, an object is returned, containing:

- **id** (pubkey): The public key unique to this node
- **alias** (string): The fun alias this node will advertize (up to 32 characters)
- **color** (hex): The favorite RGB color this node will advertize (always 6 characters)

- **num_peers** (u32): The total count of peers, connected or with channels
- **num_pending_channels** (u32): The total count of channels being opened
- **num_active_channels** (u32): The total count of channels in normal state
- **num_inactive_channels** (u32): The total count of channels waiting for opening or closing transactions to be mined
- **version** (string): Identifies what bugs you are running into
- **lightning-dir** (string): Identifies where you can find the configuration and other related files
- **blockheight** (u32): The highest block height we've learned
- **network** (string): represents the type of network on the node are working (e.g: `bitcoin`, `testnet`, or `regtest`)
- **fees_collected_msat** (msat): Total routing fees collected by this node
- **our_features** (object, optional): Our BOLT #9 feature bits (as hexstring) for various contexts:
  - **init** (hex): features (incl. globalfeatures) in our init message, these also restrict what we offer in open_channel or accept in accept_channel
  - **node** (hex): features in our node_announcement message
  - **channel** (hex): negotiated channel features we (as channel initiator) publish in the channel_announcement message
  - **invoice** (hex): features in our BOLT11 invoices
- **address** (array of objects, optional): The addresses we announce to the world:
  - **type** (string): Type of connection (one of "dns", "ipv4", "ipv6", "torv2", "torv3", "websocket")
  - **port** (u16): port number

  If **type** is "dns", "ipv4", "ipv6", "torv2" or "torv3":

  - **address** (string): address in expected format for **type**
- **binding** (array of objects, optional): The addresses we are listening on:
  - **type** (string): Type of connection (one of "local socket", "ipv4", "ipv6", "torv2", "torv3")
  - **address** (string, optional): address in expected format for **type**
  - **port** (u16, optional): port number
  - **socket** (string, optional): socket filename (only if **type** is "local socket")

The following warnings may also be returned:

- **warning_bitcoind_sync**: Bitcoind is not up-to-date with network.
- **warning_lightningd_sync**: Lightningd is still loading latest blocks from bitcoind.

On failure, one of the following error codes may be returned:

- -32602: Error in given parameters or some error happened during the command process.

47.5 EXAMPLE JSON RESPONSE

```
{
   "id": "02bf811f7571754f0b51e6d41a8885f5561041a7b14fac093e4cffb95749de1a8d",
   "alias": "SLICKERGOPHER",
   "color": "02bf81",
   "num_peers": 0,
   "num_pending_channels": 0,
   "num_active_channels": 0,
   "num_inactive_channels": 0,
   "address": [
      {
         "type": "torv3",
         "address": "fp463inc4w3lamhhduytrwdwq6q6uzugtaeapylqfc43agrdnnqsheyd.onion",
         "port": 9736
      },
      {
         "type": "torv3",
         "address": "ifnntp5ak4homxrti2fp6ckyllaqcike447ilqfrgdw64ayrmkyashid.onion",
         "port": 9736
      }
   ],
   "binding": [
      {
         "type": "ipv4",
         "address": "127.0.0.1",
         "port": 9736
      }
   ],
   "version": "v0.10.2",
   "blockheight": 724302,
   "network": "bitcoin",
   "msatoshi_fees_collected": 0,
   "fees_collected_msat": "0msat",
   "lightning-dir": "/media/vincent/Maxtor/C-lightning/node/bitcoin"
   "our_features": {
      "init": "8828226aa2",
      "node": "80008828226aa2",
      "channel": "",
      "invoice": "20024200"
   }
}
```

47.6 AUTHOR

Vincenzo Palazzo <vincenzo.palazzo@protonmail.com> wrote the initial version of this man page, but many others did the hard work of actually implementing this rpc command.

47.7 SEE ALSO

lightning-connect(7), lightning-fundchannel(7), lightning-listconfigs(7).

47.8 RESOURCES

Main web site: https://github.com/ElementsProject/lightning *comment*: # (SHA256STAMP:0e54af449933b833f2e74bab9fde46096a79d69b4d958a548c7c0b7cc5654e99)

CHAPTER 48

lightning-getlog – Command to show logs.

48.1 SYNOPSIS

getlog [*level*]

48.2 DESCRIPTION

The **getlog** the RPC command to show logs, with optional log *level*.

- *level*: A string that represents the log level (*broken*, *unusual*, *info*, *debug*, or *io*). The default is *info*.

48.3 EXAMPLE JSON REQUEST

```
{
  "id": 82,
  "method": "getlog",
  "params": {
    "level": "debug"
  }
}
```

48.4 RETURN VALUE

On success, an object is returned, containing:

- **created_at** (string): UNIX timestamp with 9 decimal places, when logging was initialized
- **bytes_used** (u32): The number of bytes used by logging records

- **bytes_max** (u32): The bytes_used values at which records will be trimmed
- **log** (array of objects):
  - **type** (string) (one of "SKIPPED", "BROKEN", "UNUSUAL", "INFO", "DEBUG", "IO_IN", "IO_OUT")

  If **type** is "SKIPPED":

  - **num_skipped** (u32): number of unprinted log entries (deleted or below *level* parameter)

  If **type** is "BROKEN", "UNUSUAL", "INFO" or "DEBUG":

  - **time** (string): UNIX timestamp with 9 decimal places after **created_at**
  - **source** (string): The particular logbook this was found in
  - **log** (string): The actual log message
  - **node_id** (pubkey, optional): The peer this is associated with

  If **type** is "IO_IN" or "IO_OUT":

  - **time** (string): Seconds after **created_at**, with 9 decimal places
  - **source** (string): The particular logbook this was found in
  - **log** (string): The associated log message
  - **data** (hex): The IO which occurred
  - **node_id** (pubkey, optional): The peer this is associated with

On failure, one of the following error codes may be returned:

- -32602: Error in given parameters.

48.5 EXAMPLE JSON RESPONSE

```
{
   "created_at": "1598192543.820753463",
   "bytes_used": 89285843,
   "bytes_max": 104857600,
   "log": [
      {
         "type": "SKIPPED",
         "num_skipped": 45
      },
      {
         "type": "INFO",
         "time": "0.453627568",
         "source": "plugin-autopilot.py",
         "log": "RPC method 'autopilot-run-once' does not have a docstring."
      }
   ]
}
```

48.6 AUTHOR

Vincenzo Palazzo <vincenzo.palazzo@protonmail.com> wrote the initial version of this man page, but many others did the hard work of actually implementing this rpc command.

48.7 RESOURCES

Main web site: https://github.com/ElementsProject/lightning *comment*: # (SHA256STAMP:0f6e346c57e59aa8ebe0aee9bcb7ded6f66776752e55c4c125f4a80d98cf90fd)

CHAPTER 49

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

49.1 SYNOPSIS

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

49.2 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* non-negative 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 non-negative 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.

49.3 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.

49.4 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.

49.5 RETURN VALUE

On success, an object containing **route** is returned. It is an array of objects, where each object contains:

- **id** (pubkey): The node at the end of this hop
- **channel** (short_channel_id): The channel joining these nodes
- **direction** (u32): 0 if this channel is traversed from lesser to greater **id**, otherwise 1
- **amount_msat** (msat): The amount expected by the node at the end of this hop
- **delay** (u32): The total CLTV expected by the node at the end of this hop
- **style** (string): The features understood by the destination node (always "tlv")

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 (assuming the first hop is free). The first *delay* is the very worst case timeout for the payment failure, in blocks.

49.6 AUTHOR

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

49.7 SEE ALSO

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

49.8 RESOURCES

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

CHAPTER 50

lightning-help – Command to return all information about RPC commands.

50.1 SYNOPSIS

help [*command*]

50.2 DESCRIPTION

The **help** is a RPC command which is possible consult all information about the RPC commands, or a specific command if *command* is given.

Note that the lightning-cli(1) tool will prefer to list a man page when a specific *command* is specified, and will only return the JSON if the man page is not found.

50.3 EXAMPLE JSON REQUEST

```
{
  "id": 82,
  "method": "help",
  "params": {}
}
```

50.4 RETURN VALUE

On success, an object is returned, containing:

- **help** (array of objects):
  - **command** (string): the command

  - **category** (string): the category for this command (useful for grouping)
  - **description** (string): a one-line description of the purpose of this command
  - **verbose** (string): a full description of this command (including whether it's deprecated)

On failure, one of the following error codes may be returned:

- -32602: Error in given parameters.

50.5 EXAMPLE JSON RESPONSE

```
{
    "help": [
      {
        "command": "autocleaninvoice [cycle_seconds] [expired_by]",
        "category": "plugin",
        "description": "Set up autoclean of expired invoices. ",
        "verbose": "Perform cleanup every {cycle_seconds} (default 3600), or disable␣
↪autoclean if 0. Clean up expired invoices that have expired for {expired_by}␣
↪seconds (default 86400). "
      }
    ]
}
```

50.6 AUTHOR

Vincenzo Palazzo <vincenzo.palazzo@protonmail.com> wrote the initial version of this man page, but many others did the hard work of actually implementing this rpc command.

50.7 RESOURCES

Main web site: https://github.com/ElementsProject/lightning *comment*: # (SHA256STAMP:48f9ef4d1d73aa13ebd1ffa37107111c35c1a197bbcf00f52c5149847ca57ac1)

CHAPTER 51

lightning-hsmtool – Tool for working with software HSM secrets of lightningd

51.1 SYNOPSIS

```
lightning-hsmtool method [ARGUMENTS]...
```

51.2 DESCRIPTION

lightning-hsmtool performs various operations on the `hsm_secret` file used by the software HSM component of **lightningd**.

This can be used to encrypt and decrypt the `hsm_secret` file, as well as derive secrets used in channel commitments.

51.3 METHODS

encrypt *hsm_secret password*

Encrypt the `hsm_secret` file so that it can only be decrypted at **lightningd** startup. You must give the option **–encrypted-hsm** to **lightningd**. The password of the `hsm_secret` file will be asked whenever you start **lightningd**.

decrypt *hsm_secret password*

Decrypt the `hsm_secret` file that was encrypted with the **encrypt** method.

dumpcommitments *node_id channel_dbid depth hsm_secret* [*password*]

Show the per-commitment secret and point of up to *depth* commitments, of the specified channel with the specified peer, identified by the channel database index. Specify *password* if the `hsm_secret` is encrypted.

guesstoremote *p2wpkh node_id max_channel_dbid hsm_secret* [*password*]

Brute-force the private key to our funds from a remote unilateral close of a channel, in a case where we have lost all database data except for our `hsm_secret`. The peer must be the one to close the channel (and the funds will remain unrecoverable until the channel is closed). *max_channel_dbid* is your own guess on what the *channel_dbid* was, or at least the maximum possible value, and is usually no greater than the number of channels that the node has ever had. Specify *password* if the `hsm_secret` is encrypted.

generatehsm *hsm_secret_path* Generates a new hsm_secret using BIP39.

checkhsm *hsm_secret_path* Checks that hsm_secret matches a BIP39 pass phrase.

dumponchaindescriptors *hsm_secret* [*password*] [*network*] Dump output descriptors for our onchain wallet. The descriptors can be used by external services to be able to generate addresses for our onchain wallet. (for example on `bitcoind` using the `importmulti` or `importdescriptors` RPC calls) We need the path to the hsm_secret containing the wallet seed, and an optional (skip using `""`) password if it was encrypted. To generate descriptors using testnet master keys, you may specify *testnet* as the last parameter. By default, mainnet-encoded keys are generated.

51.4 BUGS

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

51.5 AUTHOR

ZmnSCPxj <ZmnSCPxj@protonmail.com> wrote the initial version of this man page, but many others did the hard work of actually implementing **lightning-hsmtool**.

51.6 SEE ALSO

lightningd(8), lightningd-config(5)

51.7 RESOURCES

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

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

CHAPTER 52

lightning-invoice – Command for accepting payments

52.1 SYNOPSIS

invoice *amount_msat label description* [*expiry*] [*fallbacks*] [*preimage*] [*exposeprivatechannels*] [*cltv*] [*deschashonly*]

52.2 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 *amount_msat* parameter can be the string "any", which creates an invoice that can be paid with any amount. Otherwise it is a positive value 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 (unless *deschashonly* is true as described below). It must be UTF-8, and cannot use *\u* JSON escape codes.

The *expiry* is optionally the time the invoice is valid for, in seconds. If no value is provided the default of 604800 (1 week) 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 (or a remote alias), only those specific channels will be considered candidates, even if they are public or dead-ends.

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.

If specified, *cltv* sets the *min_final_cltv_expiry* for the invoice. Otherwise, it's set to the parameter **cltv-final**.

If *deschashonly* is true (default false), then the bolt11 returned contains a hash of the *description*, rather than the *description* itself: this allows much longer descriptions, but they must be communicated via some other mechanism.

52.3 RETURN VALUE

On success, an object is returned, containing:

- **bolt11** (string): the bolt11 string
- **payment_hash** (hash): the hash of the *payment_preimage* which will prove payment (always 64 characters)
- **payment_secret** (secret): the *payment_secret* to place in the onion (always 64 characters)
- **expires_at** (u64): UNIX timestamp of when invoice expires

The following warnings may also be returned:

- **warning_capacity**: even using all possible channels, there's not enough incoming capacity to pay this invoice.
- **warning_offline**: there would be enough incoming capacity, but some channels are offline, so there isn't.
- **warning_deadends**: there would be enough incoming capacity, but some channels are dead-ends (no other public channels from those peers), so there isn't.
- **warning_private_unused**: there would be enough incoming capacity, but some channels are unannounced and *exposeprivatechannels* is *false*, so there isn't.
- **warning_mpp**: there is sufficient capacity, but not in a single channel, so the payer will have to use multi-part payments.

On failure, an error is returned and no invoice is created. If the lightning process fails before responding, the caller should use lightning-listinvoices(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.

52.4 AUTHOR

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

52.5 SEE ALSO

lightning-listinvoices(7), lightning-delinvoice(7), lightning-pay(7).

52.6 RESOURCES

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

CHAPTER 53

lightning-keysend – Send funds to a node without an invoice

53.1 SYNOPSIS

keysend *destination* *msatoshi* [*label*] [*maxfeepercent*] [*retry_for*] [*maxdelay*] [*exemptfee*] [*extratlvs*]

53.2 DESCRIPTION

The **keysend** RPC command attempts to find a route to the given destination, and send the specified amount to it. Unlike the `pay` RPC command the `keysend` command does not require an invoice, instead it uses the `destination` node ID, and `amount` to find a route to the specified node.

In order for the destination to be able to claim the payment, the `payment_key` is randomly generated by the sender and included in the encrypted payload for the destination. As a consequence there is not proof-of-payment, like there is with an invoice where the `payment_key` is generated on the destination, and the only way sender could have it is by sending a payment. Please ensure that this matches your use-case when using `keysend`.

`destination` is the 33 byte, hex-encoded, node ID of the node that the payment should go to. `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 `maxfeepercent` limits the money paid in fees as percentage of the total amount that is to be transferred, and defaults to *0.5*. 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. Unlike lightning-pay(7), issuing the same `keysend` commands multiple times will result in multiple payments being sent.

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.

extratlvs is an optional dictionary of additional fields to insert into the final tlv. The format is 'fieldnumber': 'hexstring'.

When using *lightning-cli*, you may skip optional parameters by using *null*. Alternatively, use **-k** option to provide parameters by name.

53.3 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*.

53.4 RETURN VALUE

On success, an object is returned, containing:

- **payment_preimage** (secret): the proof of payment: SHA256 of this **payment_hash** (always 64 characters)
- **payment_hash** (hash): the hash of the *payment_preimage* which will prove payment (always 64 characters)
- **created_at** (number): the UNIX timestamp showing when this payment was initiated
- **parts** (u32): how many attempts this took
- **amount_msat** (msat): Amount the recipient received
- **amount_sent_msat** (msat): Total amount we sent (including fees)
- **status** (string): status of payment (always "complete")
- **destination** (pubkey, optional): the final destination of the payment

The following warnings may also be returned:

- **warning_partial_completion**: Not all parts of a multi-part payment have completed

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.
- `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.

- `210`: Payment timed out without a payment in progress.

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.

53.5 AUTHOR

Christian Decker <decker@blockstream.com> is mainly responsible.

53.6 SEE ALSO

lightning-listpays(7), lightning-decodepay(7), lightning-listinvoice(7), lightning-delinvoice(7), lightning-getroute(7), lightning-invoice(7).

53.7 RESOURCES

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

CHAPTER 54

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

54.1 SYNOPSIS

listchannels [*short_channel_id*] [*source*] [*destination*]

54.2 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 a short channel id, then only known channels with a matching *short_channel_id* are returned. Otherwise, it must be null.

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

If *destination* is a node id, then only channels leading to that node id are returned.

Only one of *short_channel_id*, *source* or *destination* can be supplied. If nothing 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.

54.3 RETURN VALUE

On success, an object containing **channels** is returned. It is an array of objects, where each object contains:

- **source** (pubkey): the source node
- **destination** (pubkey): the destination node
- **short_channel_id** (short_channel_id): short channel id of channel

- **public** (boolean): true if this is announced (otherwise it must be our channel)
- **amount_msat** (msat): the total capacity of this channel (always a whole number of satoshis)
- **message_flags** (u8): as defined by BOLT #7
- **channel_flags** (u8): as defined by BOLT #7
- **active** (boolean): true unless source has disabled it, or it's a local channel and the peer is disconnected or it's still opening or closing
- **last_update** (u32): UNIX timestamp on the last channel_update from *source*
- **base_fee_millisatoshi** (u32): Base fee changed by *source* to use this channel
- **fee_per_millionth** (u32): Proportional fee changed by *source* to use this channel, in parts-per-million
- **delay** (u32): The number of blocks delay required by *source* to use this channel
- **htlc_minimum_msat** (msat): The smallest payment *source* will allow via this channel
- **features** (hex): BOLT #9 features bitmap for this channel
- **htlc_maximum_msat** (msat, optional): The largest payment *source* will allow via this channel

If one of *short_channel_id*, *source* or *destination* 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.

54.4 AUTHOR

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

54.5 SEE ALSO

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

54.6 RESOURCES

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

Lightning RFC site

- BOLT #7: https://github.com/lightning/bolts/blob/master/07-routing-gossip.md

CHAPTER 55

lightning-listconfigs – Command to list all configuration options.

55.1 SYNOPSIS

listconfigs [*config*]

55.2 DESCRIPTION

config (optional) is a configuration option name, or "plugin" to show plugin options

The **listconfigs** RPC command to list all configuration options, or with *config* only a selection.

The returned values reflect the current configuration, including showing default values (`dev-` options are not shown).

55.3 EXAMPLE JSON REQUEST

```
{
  "id": 82,
  "method": "listconfigs",
  "params": {
    "config": "network"
  }
}
```

55.4 RETURN VALUE

On success, an object is returned, containing:

- **# version** (string, optional): Special field indicating the current version

- **plugins** (array of objects, optional):
  - **path** (string): Full path of the plugin
  - **name** (string): short name of the plugin
  - **options** (object, optional): Specific options set for this plugin:
- **important-plugins** (array of objects, optional):
  - **path** (string): Full path of the plugin
  - **name** (string): short name of the plugin
  - **options** (object, optional): Specific options set for this plugin:
- **conf** (string, optional): `conf` field from cmdline, or default
- **lightning-dir** (string, optional): `lightning-dir` field from config or cmdline, or default
- **network** (string, optional): `network` field from config or cmdline, or default
- **allow-deprecated-apis** (boolean, optional): `allow-deprecated-apis` field from config or cmdline, or default
- **rpc-file** (string, optional): `rpc-file` field from config or cmdline, or default
- **disable-plugin** (array of strings, optional):
  - `disable-plugin` field from config or cmdline
- **bookkeeper-dir** (string, optional): `bookkeeper-dir` field from config or cmdline, or default
- **bookkeeper-db** (string, optional): `bookkeeper-db` field from config or cmdline, or default
- **always-use-proxy** (boolean, optional): `always-use-proxy` field from config or cmdline, or default
- **daemon** (boolean, optional): `daemon` field from config or cmdline, or default
- **wallet** (string, optional): `wallet` field from config or cmdline, or default
- **large-channels** (boolean, optional): `large-channels` field from config or cmdline, or default
- **experimental-dual-fund** (boolean, optional): `experimental-dual-fund` field from config or cmdline, or default
- **experimental-onion-messages** (boolean, optional): `experimental-onion-messages` field from config or cmdline, or default
- **experimental-offers** (boolean, optional): `experimental-offers` field from config or cmdline, or default
- **experimental-shutdown-wrong-funding** (boolean, optional): `experimental-shutdown-wrong-funding` field from config or cmdline, or default
- **experimental-websocket-port** (u16, optional): `experimental-websocket-port` field from config or cmdline, or default
- **database-upgrade** (boolean, optional): `database-upgrade` field from config or cmdline
- **rgb** (hex, optional): `rgb` field from config or cmdline, or default (always 6 characters)
- **alias** (string, optional): `alias` field from config or cmdline, or default
- **pid-file** (string, optional): `pid-file` field from config or cmdline, or default
- **ignore-fee-limits** (boolean, optional): `ignore-fee-limits` field from config or cmdline, or default
- **watchtime-blocks** (u32, optional): `watchtime-blocks` field from config or cmdline, or default
- **max-locktime-blocks** (u32, optional): `max-locktime-blocks` field from config or cmdline, or default

- **funding-confirms** (u32, optional): `funding-confirms` field from config or cmdline, or default
- **cltv-delta** (u32, optional): `cltv-delta` field from config or cmdline, or default
- **cltv-final** (u32, optional): `cltv-final` field from config or cmdline, or default
- **commit-time** (u32, optional): `commit-time` field from config or cmdline, or default
- **fee-base** (u32, optional): `fee-base` field from config or cmdline, or default
- **rescan** (integer, optional): `rescan` field from config or cmdline, or default
- **fee-per-satoshi** (u32, optional): `fee-per-satoshi` field from config or cmdline, or default
- **max-concurrent-htlcs** (u32, optional): `max-concurrent-htlcs` field from config or cmdline, or default
- **htlc-minimum-msat** (msat, optional): `htlc-minimum-msat` field from config or cmdline, or default
- **htlc-maximum-msat** (msat, optional): `htlc-maximum-msat` field from config or cmdline, or default
- **max-dust-htlc-exposure-msat** (msat, optional): `max-dust-htlc-exposure-mast` field from config or cmdline, or default
- **min-capacity-sat** (u64, optional): `min-capacity-sat` field from config or cmdline, or default
- **addr** (string, optional): `addr` field from config or cmdline (can be more than one)
- **announce-addr** (string, optional): `announce-addr` field from config or cmdline (can be more than one)
- **bind-addr** (string, optional): `bind-addr` field from config or cmdline (can be more than one)
- **offline** (boolean, optional): `true` if `offline` was set in config or cmdline
- **autolisten** (boolean, optional): `autolisten` field from config or cmdline, or default
- **proxy** (string, optional): `proxy` field from config or cmdline, or default
- **disable-dns** (boolean, optional): `true` if `disable-dns` was set in config or cmdline
- **disable-ip-discovery** (boolean, optional): `true` if `disable-ip-discovery` was set in config or cmdline
- **encrypted-hsm** (boolean, optional): `true` if `encrypted-hsm` was set in config or cmdline
- **rpc-file-mode** (string, optional): `rpc-file-mode` field from config or cmdline, or default
- **log-level** (string, optional): `log-level` field from config or cmdline, or default
- **log-prefix** (string, optional): `log-prefix` field from config or cmdline, or default
- **log-file** (string, optional): `log-file` field from config or cmdline, or default
- **log-timestamps** (boolean, optional): `log-timestamps` field from config or cmdline, or default
- **force-feerates** (string, optional): force-feerate configuration setting, if any
- **subdaemon** (string, optional): `subdaemon` fields from config or cmdline if any (can be more than one)
- **fetchinvoice-noconnect** (boolean, optional): `fetchinvoice-noconnect` fields from config or cmdline, or default
- **accept-htlc-tlv-types** (string, optional): `accept-extra-tlvs-type` fields from config or cmdline, or not present
- **tor-service-password** (string, optional): `tor-service-password` field from config or cmdline, if any
- **dev-allowdustreserve** (boolean, optional): Whether we allow setting dust reserves

On failure, one of the following error codes may be returned:

- -32602: Error in given parameters or field with *config* name doesn't exist.

55.5 EXAMPLE JSON RESPONSE

```
{
   "# version": "v0.9.0-1",
   "lightning-dir": "/media/vincent/Maxtor/sanboxTestWrapperRPC/lightning_dir_dev",
   "network": "testnet",
   "allow-deprecated-apis": true,
   "rpc-file": "lightning-rpc",
   "plugins": [
      {
         "path": "/home/vincent/Github/plugins/sauron/sauron.py",
         "name": "sauron.py",
         "options": {
            "sauron-api-endpoint": "http://blockstream.info/testnet/api/",
            "sauron-tor-proxy": ""
         }
      },
      {
         "path": "/home/vincent/Github/reckless/reckless.py",
         "name": "reckless.py"
      }
   ],
   "important-plugins": [
      {
         "path": "/home/vincent/Github/lightning/lightningd/../plugins/autoclean",
         "name": "autoclean",
         "options": {
            "autocleaninvoice-cycle": null,
            "autocleaninvoice-expired-by": null
         }
      },
      {
         "path": "/home/vincent/Github/lightning/lightningd/../plugins/fundchannel",
         "name": "fundchannel"
      },
      {
         "path": "/home/vincent/Github/lightning/lightningd/../plugins/keysend",
         "name": "keysend"
      },
      {
         "path": "/home/vincent/Github/lightning/lightningd/../plugins/pay",
         "name": "pay",
         "options": {
            "disable-mpp": false
         }
      }
   ],
   "important-plugin": "/home/vincent/Github/lightning/lightningd/../plugins/autoclean
↪",
   "important-plugin": "/home/vincent/Github/lightning/lightningd/../plugins/
↪fundchannel",
   "important-plugin": "/home/vincent/Github/lightning/lightningd/../plugins/keysend",
   "important-plugin": "/home/vincent/Github/lightning/lightningd/../plugins/pay",
   "plugin": "/home/vincent/Github/plugins/sauron/sauron.py",
   "plugin": "/home/vincent/Github/reckless/reckless.py",
   "disable-plugin": [
      "bcli"
```

(continues on next page)

(continued from previous page)

```
  ],
  "always-use-proxy": false,
  "daemon": "false",
  "wallet": "sqlite3:///media/vincent/Maxtor/sanboxTestWrapperRPC/lightning_dir_dev/
↪testnet/lightningd.sqlite3",
  "wumbo": false,
  "wumbo": false,
  "rgb": "03ad98",
  "alias": "BRUCEWAYN-TES-DEV",
  "pid-file": "/media/vincent/Maxtor/sanboxTestWrapperRPC/lightning_dir_dev/
↪lightningd-testne...",
  "ignore-fee-limits": true,
  "watchtime-blocks": 6,
  "max-locktime-blocks": 2016,
  "funding-confirms": 1,
  "commit-fee-min": 0,
  "commit-fee-max": 0,
  "cltv-delta": 6,
  "cltv-final": 10,
  "commit-time": 10,
  "fee-base": 1,
  "rescan": 30,
  "fee-per-satoshi": 10,
  "max-concurrent-htlcs": 483,
  "min-capacity-sat": 10000,
  "addr": "autotor:127.0.0.1:9051",
  "bind-addr": "127.0.0.1:9735",
  "announce-addr": "fp463inc4w3lamhhduytrwdwq6q6uzugtaeapylqfc43agrdnnqsheyd.
↪onion:9735",
  "offline": "false",
  "autolisten": true,
  "proxy": "127.0.0.1:9050",
  "disable-dns": "false",
  "encrypted-hsm": false,
  "rpc-file-mode": "0600",
  "log-level": "DEBUG",
  "log-prefix": "lightningd",
}
```

55.6 AUTHOR

Vincenzo Palazzo <vincenzo.palazzo@protonmail.com> wrote the initial version of this man page, but many others did the hard work of actually implementing this rpc command.

55.7 SEE ALSO

lightning-getinfo(7), lightningd-config(5)

55.8 RESOURCES

Main web site: https://github.com/ElementsProject/lightning *comment*: # (SHA256STAMP:5871ac751654339ed65ab905d61f0bc3afbb8576a33a5c4e9a73d2084f438582)

CHAPTER 56

lightning-listdatastore – Command for listing (plugin) data

56.1 SYNOPSIS

listdatastore [*key*]

56.2 DESCRIPTION

The **listdatastore** RPC command allows plugins to fetch data which was stored in the Core Lightning database.

All immediate children of the *key* (or root children) are returned: a *key* with children won't have a *hex* or *generation* entry.

56.3 RETURN VALUE

On success, an object containing **datastore** is returned. It is an array of objects, where each object contains:

- **key** (array of strings):
  - Part of the key added to the datastore
- **generation** (u64, optional): The number of times this has been updated
- **hex** (hex, optional): The hex data from the datastore
- **string** (string, optional): The data as a string, if it's valid utf-8

The following error codes may occur:

- -32602: invalid parameters.

56.4 AUTHOR

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

56.5 SEE ALSO

lightning-datastore(7), lightning-deldatastore(7)

56.6 RESOURCES

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

CHAPTER 57

lightning-listforwards – Command showing all htlcs and their information

57.1 SYNOPSIS

listforwards [*status*] [*in_channel*] [*out_channel*]

57.2 DESCRIPTION

The **listforwards** RPC command displays all htlcs that have been attempted to be forwarded by the Core Lightning node.

If *status* is specified, then only the forwards with the given status are returned. *status* can be either *offered* or *settled* or *failed* or *local_failed*

If *in_channel* or *out_channel* is specified, then only the matching forwards on the given in/out channel are returned.

57.3 RETURN VALUE

On success, an object containing **forwards** is returned. It is an array of objects, where each object contains:

- **in_channel** (short_channel_id): the channel that received the HTLC
- **in_msat** (msat): the value of the incoming HTLC
- **status** (string): still ongoing, completed, failed locally, or failed after forwarding (one of "offered", "settled", "local_failed", "failed")
- **received_time** (number): the UNIX timestamp when this was received
- **in_htlc_id** (u64, optional): the unique HTLC id the sender gave this (not present if incoming channel was closed before ugprade to v22.11)
- **out_channel** (short_channel_id, optional): the channel that the HTLC (trying to) forward to

- **out_htlc_id** (u64, optional): the unique HTLC id we gave this when sending (may be missing even if out_channel is present, for old forwards before v22.11)
- **style** (string, optional): Either a legacy onion format or a modern tlv format (one of "legacy", "tlv")

If **out_msat** is present:

- **fee_msat** (msat): the amount this paid in fees
- **out_msat** (msat): the amount we sent out the *out_channel*

If **status** is "settled" or "failed":

- **resolved_time** (number): the UNIX timestamp when this was resolved

If **status** is "local_failed" or "failed":

- **failcode** (u32, optional): the numeric onion code returned
- **failreason** (string, optional): the name of the onion code returned

57.4 AUTHOR

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

57.5 SEE ALSO

lightning-autoclean-status(7), lightning-getinfo(7)

57.6 RESOURCES

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

CHAPTER 58

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

58.1 SYNOPSIS

listfunds [*spent*]

58.2 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.

spent is a boolean: if true, then the *outputs* will include spent outputs in addition to the unspent ones. Default is false.

58.3 RETURN VALUE

On success, an object is returned, containing:

- **outputs** (array of objects):
  - **txid** (txid): the ID of the spendable transaction
  - **output** (u32): the index within *txid*
  - **amount_msat** (msat): the amount of the output
  - **scriptpubkey** (hex): the scriptPubkey of the output
  - **status** (string) (one of "unconfirmed", "confirmed", "spent", "immature")
  - **reserved** (boolean): whether this UTXO is currently reserved for an in-flight tx
  - **address** (string, optional): the bitcoin address of the output

  - **redeemscript** (hex, optional): the redeemscript, only if it's p2sh-wrapped

  If **status** is "confirmed":

  - **blockheight** (u32): Block height where it was confirmed

  If **reserved** is "true":

  - **reserved_to_block** (u32): Block height where reservation will expire

- **channels** (array of objects):
  - **peer_id** (pubkey): the peer with which the channel is opened
  - **our_amount_msat** (msat): available satoshis on our node's end of the channel
  - **amount_msat** (msat): total channel value
  - **funding_txid** (txid): funding transaction id
  - **funding_output** (u32): the 0-based index of the output in the funding transaction
  - **connected** (boolean): whether the channel peer is connected
  - **state** (string): the channel state, in particular "CHANNELD_NORMAL" means the channel can be used normally (one of "OPENINGD", "CHANNELD_AWAITING_LOCKIN", "CHANNELD_NORMAL", "CHANNELD_SHUTTING_DOWN", "CLOSINGD_SIGEXCHANGE", "CLOSINGD_COMPLETE", "AWAITING_UNILATERAL", "FUNDING_SPEND_SEEN", "ONCHAIN", "DUALOPEND_OPEN_INIT", "DUALOPEND_AWAITING_LOCKIN")

  If **state** is "CHANNELD_NORMAL":

  - **short_channel_id** (short_channel_id): short channel id of channel

  If **state** is "CHANNELD_SHUTTING_DOWN", "CLOSINGD_SIGEXCHANGE", "CLOSINGD_COMPLETE", "AWAITING_UNILATERAL", "FUNDING_SPEND_SEEN" or "ONCHAIN":

  - **short_channel_id** (short_channel_id, optional): short channel id of channel (only if funding reached lockin depth before closing)

58.4 AUTHOR

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

58.5 SEE ALSO

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

58.6 RESOURCES

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

CHAPTER 59

lightning-listhtlcs – Command for querying HTLCs

59.1 SYNOPSIS

listhtlcs [*id*]

59.2 DESCRIPTION

The **listhtlcs** RPC command gets all HTLCs (which, generally, we remember for as long as a channel is open, even if they've completed long ago). If given a short channel id (e.g. 1x2x3) or full 64-byte hex channel id, it will only list htlcs for that channel (which must be known).

59.3 RETURN VALUE

On success, an object containing **htlcs** is returned. It is an array of objects, where each object contains:

- **short_channel_id** (short_channel_id): the channel that contains/contained the HTLC
- **id** (u64): the unique, incrementing HTLC id the creator gave this
- **expiry** (u32): the block number where this HTLC expires/expired
- **amount_msat** (msat): the value of the HTLC
- **direction** (string): out if we offered this to the peer, in if they offered it (one of "out", "in")
- **payment_hash** (hex): payment hash sought by HTLC (always 64 characters)
- **state** (string): The first 10 states are for `in`, the next 10 are for `out`. (one of "SENT_ADD_HTLC", "SENT_ADD_COMMIT", "RCVD_ADD_REVOCATION", "RCVD_ADD_ACK_COMMIT", "SENT_ADD_ACK_REVOCATION", "RCVD_REMOVE_HTLC", "RCVD_REMOVE_COMMIT", "SENT_REMOVE_REVOCATION", "SENT_REMOVE_ACK_COMMIT", "RCVD_REMOVE_ACK_REVOCATION", "RCVD_ADD_HTLC", "RCVD_ADD_COMMIT",

"SENT_ADD_REVOCATION", "SENT_ADD_ACK_COMMIT", "RCVD_ADD_ACK_REVOCATION", "SENT_REMOVE_HTLC", "SENT_REMOVE_COMMIT", "RCVD_REMOVE_REVOCATION", "RCVD_REMOVE_ACK_COMMIT", "SENT_REMOVE_ACK_REVOCATION")

59.4 AUTHOR

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

59.5 SEE ALSO

lightning-listforwards(7)

59.6 RESOURCES

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

CHAPTER 60

lightning-listinvoices – Command for querying invoice status

60.1 SYNOPSIS

listinvoices [*label*] [*invstring*] [*payment_hash*] [*offer_id*]

60.2 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.

A specific invoice can be queried by providing either the `label` provided when creating the invoice, the `invstring` string representing the invoice, the `payment_hash` of the invoice, or the local `offer_id` this invoice was issued for. Only one of the query parameters can be used at once.

60.3 RETURN VALUE

On success, an object containing **invoices** is returned. It is an array of objects, where each object contains:

- **label** (string): unique label supplied at invoice creation
- **payment_hash** (hash): the hash of the *payment_preimage* which will prove payment (always 64 characters)
- **status** (string): Whether it's paid, unpaid or unpayable (one of "unpaid", "paid", "expired")
- **expires_at** (u64): UNIX timestamp of when it will become / became unpayable
- **description** (string, optional): description used in the invoice
- **amount_msat** (msat, optional): the amount required to pay this invoice
- **bolt11** (string, optional): the BOLT11 string (always present unless *bolt12* is)
- **bolt12** (string, optional): the BOLT12 string (always present unless *bolt11* is)

- **local_offer_id** (hex, optional): the *id* of our offer which created this invoice (**experimental-offers** only). (always 64 characters)
- **invreq_payer_note** (string, optional): the optional *invreq_payer_note* from invoice_request which created this invoice (**experimental-offers** only).

If **status** is "paid":

- **pay_index** (u64): Unique incrementing index for this payment
- **amount_received_msat** (msat): the amount actually received (could be slightly greater than *amount_msat*, since clients may overpay)
- **paid_at** (u64): UNIX timestamp of when it was paid
- **payment_preimage** (secret): proof of payment (always 64 characters)

60.4 AUTHOR

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

60.5 SEE ALSO

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

60.6 RESOURCES

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

CHAPTER 61

lightning-listnodes – Command to get the list of nodes in the known network.

61.1 SYNOPSIS

listnodes [*id*]

61.2 DESCRIPTION

The **listnodes** command returns nodes the node has learned about via gossip messages, or a single one if the node *id* was specified.

61.3 EXAMPLE JSON REQUEST

```
{
  "id": 82,
  "method": "listnodes",
  "params": {
    "id": "02e29856dab8ddd9044c18486e4cab79ec717b490447af2d4831e290e48d57638a"
  }
}
```

61.4 RETURN VALUE

On success, an object containing **nodes** is returned. It is an array of objects, where each object contains:

- **nodeid** (pubkey): the public key of the node
- **last_timestamp** (u32, optional): A node_announcement has been received for this node (UNIX timestamp)

If **last_timestamp** is present:

- **alias** (string): The fun alias this node advertized (up to 32 characters)
- **color** (hex): The favorite RGB color this node advertized (always 6 characters)
- **features** (hex): BOLT #9 features bitmap this node advertized
- **addresses** (array of objects): The addresses this node advertized:
  - **type** (string): Type of connection (one of "dns", "ipv4", "ipv6", "torv2", "torv3", "websocket")
  - **port** (u16): port number

  If **type** is "dns", "ipv4", "ipv6", "torv2" or "torv3":

  - **address** (string): address in expected format for **type**

If **option_will_fund** is present:

- **option_will_fund** (object):
  - **lease_fee_base_msat** (msat): the fixed fee for a lease (whole number of satoshis)
  - **lease_fee_basis** (u32): the proportional fee in basis points (parts per 10,000) for a lease
  - **funding_weight** (u32): the onchain weight you'll have to pay for a lease
  - **channel_fee_max_base_msat** (msat): the maximum base routing fee this node will charge during the lease
  - **channel_fee_max_proportional_thousandths** (u32): the maximum proportional routing fee this node will charge during the lease (in thousandths, not millionths like channel_update)
  - **compact_lease** (hex): the lease as represented in the node_announcement

On failure, one of the following error codes may be returned:

- -32602: Error in given parameters.

61.5 EXAMPLE JSON RESPONSE

```
{
   "nodes": [
      {
         "nodeid": "02e29856dab8ddd9044c14586e4cab79ec717b490447af2d4831e290e48d58638a
↪",
         "alias": "some_alias",
         "color": "68f442",
         "last_timestamp": 1597213741,
         "features": "02a2a1",
         "addresses": [
            {
               "type": "ipv4",
               "address": "zzz.yy.xx.xx",
               "port": 9735
            }
         ]
      }
    ]
}
```

61.6 AUTHOR

Vincenzo Palazzo <vincenzo.palazzo@protonmail.com> wrote the initial version of this man page, but many others did the hard work of actually implementing this rpc command.

61.7 SEE ALSO

FIXME:

61.8 RESOURCES

Main web site: https://github.com/ElementsProject/lightning *comment*: # (SHA256STAMP:7f1378c1376ade1c9912c8eef3ebc77b13cbc5194ee813f8f1b4e0061338e0bb)

CHAPTER 62

lightning-listoffers – Command for listing offers

62.1 SYNOPSIS

(WARNING: experimental-offers only)

listoffers [*offer_id*] [*active_only*]

62.2 DESCRIPTION

The **listoffers** RPC command list all offers, or with `offer_id`, only the offer with that offer_id (if it exists). If `active_only` is set and is true, only offers with `active` true are returned.

62.3 EXAMPLE JSON REQUEST

```
{
  "id": 82,
  "method": "listoffers",
  "params": {
        "active_only": false
  }
}
```

62.4 RETURN VALUE

On success, an object containing **offers** is returned. It is an array of objects, where each object contains:

- **offer_id** (hex): the id of this offer (merkle hash of non-signature fields) (always 64 characters)
- **active** (boolean): whether this can still be used

- **single_use** (boolean): whether this expires as soon as it's paid
- **bolt12** (string): the bolt12 encoding of the offer
- **used** (boolean): True if an associated invoice has been paid
- **label** (string, optional): the (optional) user-specified label

62.5 EXAMPLE JSON RESPONSE

```
{
  "offers": [
    {
      "offer_id": "053a5c566fbea2681a5ff9c05a913da23e45b95d09ef5bd25d7d408f23da7084",
      "active": true,
      "single_use": false,
      "bolt12":
↪"lno1qgsqvgnwgcg35z6ee2h3yczraddm72xrfua9uve2rlrm9deu7xyfzrcgqvqcdgq2z9pk7enxv4jjqen0wgs8yatnw3ujz8
↪",
      "used": false
    },
    {
      "offer_id": "3247d3597fec19e362ca683416a48a0f76a44c1600725a7ee1936548feadacca",
      "active": true,
      "single_use": false,
      "bolt12":
↪"lno1qgsqvgnwgcg35z6ee2h3yczraddm72xrfua9uve2rlrm9deu7xyfzrcxqd24x3qgqgqlgzs3gdhkven9v5sxvmmjype82u
↪",
      "used": true
    }
  ]
}
```

62.6 AUTHOR

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

62.7 SEE ALSO

lightning-offer(7), lightning-offerout(7), lightning-listoffers(7).

62.8 RESOURCES

Main web site: https://github.com/ElementsProject/lightning *comment*: # (SHA256STAMP:985a6bae4b0a1702cd02998859c8072eee44b219c15294af4f4078465531c8c9)

CHAPTER 63

lightning-listpays – Command for querying payment status

63.1 SYNOPSIS

listpays [*bolt11*] [*payment_hash*] [*status*]

63.2 DESCRIPTION

The **listpay** RPC command gets the status of all *pay* commands, or a single one if either *bolt11* or *payment_hash* was specified. It is possible filter the payments also by *status*.

63.3 RETURN VALUE

On success, an object containing **pays** is returned. It is an array of objects, where each object contains:

- **payment_hash** (hex): the hash of the *payment_preimage* which will prove payment (always 64 characters)
- **status** (string): status of the payment (one of "pending", "failed", "complete")
- **created_at** (u64): the UNIX timestamp showing when this payment was initiated
- **destination** (pubkey, optional): the final destination of the payment if known
- **completed_at** (u64, optional): the UNIX timestamp showing when this payment was completed
- **label** (string, optional): the label, if given to sendpay
- **bolt11** (string, optional): the bolt11 string (if pay supplied one)
- **description** (string, optional): the description matching the bolt11 description hash (if pay supplied one)
- **bolt12** (string, optional): the bolt12 string (if supplied for pay: **experimental-offers** only).

If **status** is "complete":

- **preimage** (hex): proof of payment (always 64 characters)
- **number_of_parts** (u64, optional): the number of parts for a successful payment (only if more than one).

If **status** is "failed":

- **erroronion** (hex, optional): the error onion returned on failure, if any.

The returned array is ordered by increasing **created_at** fields.

63.4 AUTHOR

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

63.5 SEE ALSO

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

63.6 RESOURCES

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

CHAPTER 64

lightning-listpeers – Command returning data on connected lightning nodes

64.1 SYNOPSIS

listpeers [*id*] [*level*]

64.2 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.

64.3 RETURN VALUE

On success, an object containing **peers** is returned. It is an array of objects, where each object contains:

- **id** (pubkey): the public key of the peer
- **connected** (boolean): True if the peer is currently connected
- **channels** (array of objects):
  - **state** (string): the channel state, in particular "CHANNELD_NORMAL" means the channel can be used normally (one of "OPENINGD", "CHANNELD_AWAITING_LOCKIN", "CHANNELD_NORMAL", "CHANNELD_SHUTTING_DOWN", "CLOSINGD_SIGEXCHANGE", "CLOSINGD_COMPLETE", "AWAITING_UNILATERAL", "FUNDING_SPEND_SEEN", "ONCHAIN", "DUALOPEND_OPEN_INIT", "DUALOPEND_AWAITING_LOCKIN")
  - **opener** (string): Who initiated the channel (one of "local", "remote")
  - **features** (array of strings):
    - BOLT #9 features which apply to this channel (one of "option_static_remotekey", "option_anchor_outputs", "option_zeroconf")
  - **scratch_txid** (txid, optional): The txid we would use if we went onchain now
  - **feerate** (object, optional): Feerates for the current tx:
    - **perkw** (u32): Feerate per 1000 weight (i.e kSipa)
    - **perkb** (u32): Feerate per 1000 virtual bytes
  - **owner** (string, optional): The current subdaemon controlling this connection
  - **short_channel_id** (short_channel_id, optional): The short_channel_id (once locked in)
  - **channel_id** (hash, optional): The full channel_id (always 64 characters)
  - **funding_txid** (txid, optional): ID of the funding transaction
  - **funding_outnum** (u32, optional): The 0-based output number of the funding transaction which opens the channel
  - **initial_feerate** (string, optional): For inflight opens, the first feerate used to initiate the channel open
  - **last_feerate** (string, optional): For inflight opens, the most recent feerate used on the channel open
  - **next_feerate** (string, optional): For inflight opens, the next feerate we'll use for the channel open
  - **next_fee_step** (u32, optional): For inflight opens, the next feerate step we'll use for the channel open
  - **inflight** (array of objects, optional): Current candidate funding transactions (only for dual-funding):
    - **funding_txid** (txid): ID of the funding transaction
    - **funding_outnum** (u32): The 0-based output number of the funding transaction which opens the channel
    - **feerate** (string): The feerate for this funding transaction in per-1000-weight, with "kpw" appended
    - **total_funding_msat** (msat): total amount in the channel
    - **our_funding_msat** (msat): amount we have in the channel
    - **scratch_txid** (txid): The commitment transaction txid we would use if we went onchain now
  - **close_to** (hex, optional): scriptPubkey which we have to close to if we mutual close
  - **private** (boolean, optional): if False, we will not announce this channel
  - **closer** (string, optional): Who initiated the channel close (one of "local", "remote")
  - **funding** (object, optional):

    * **local_funds_msat** (msat): Amount of channel we funded
    * **remote_funds_msat** (msat): Amount of channel they funded
    * **local_msat** (msat, optional): Amount of channel we funded (deprecated)
    * **remote_msat** (msat, optional): Amount of channel they funded (deprecated)
    * **pushed_msat** (msat, optional): Amount pushed from opener to peer
    * **fee_paid_msat** (msat, optional): Amount we paid peer at open
    * **fee_rcvd_msat** (msat, optional): Amount we were paid by peer at open
  - **to_us_msat** (msat, optional): how much of channel is owed to us
  - **min_to_us_msat** (msat, optional): least amount owed to us ever
  - **max_to_us_msat** (msat, optional): most amount owed to us ever
  - **total_msat** (msat, optional): total amount in the channel
  - **fee_base_msat** (msat, optional): amount we charge to use the channel
  - **fee_proportional_millionths** (u32, optional): amount we charge to use the channel in parts-per-million
  - **dust_limit_msat** (msat, optional): minimum amount for an output on the channel transactions
  - **max_total_htlc_in_msat** (msat, optional): max amount accept in a single payment
  - **their_reserve_msat** (msat, optional): minimum we insist they keep in channel
  - **our_reserve_msat** (msat, optional): minimum they insist we keep in channel
  - **spendable_msat** (msat, optional): total we could send through channel
  - **receivable_msat** (msat, optional): total peer could send through channel
  - **minimum_htlc_in_msat** (msat, optional): the minimum amount HTLC we accept
  - **minimum_htlc_out_msat** (msat, optional): the minimum amount HTLC we will send
  - **maximum_htlc_out_msat** (msat, optional): the maximum amount HTLC we will send
  - **their_to_self_delay** (u32, optional): the number of blocks before they can take their funds if they unilateral close
  - **our_to_self_delay** (u32, optional): the number of blocks before we can take our funds if we unilateral close
  - **max_accepted_htlcs** (u32, optional): Maximum number of incoming HTLC we will accept at once
  - **alias** (object, optional):
    * **local** (short_channel_id, optional): An alias assigned by this node to this channel, used for outgoing payments
    * **remote** (short_channel_id, optional): An alias assigned by the remote node to this channel, usable in routehints and invoices
  - **state_changes** (array of objects, optional): Prior state changes:
    * **timestamp** (string): UTC timestamp of form YYYY-mm-ddTHH:MM:SS.%03dZ
    * **old_state** (string): Previous state (one of "OPENINGD", "CHANNELD_AWAITING_LOCKIN", "CHANNELD_NORMAL", "CHANNELD_SHUTTING_DOWN", "CLOSINGD_SIGEXCHANGE", "CLOSINGD_COMPLETE", "AWAITING_UNILATERAL", "FUNDING_SPEND_SEEN", "ONCHAIN", "DUALOPEND_OPEN_INIT", "DUALOPEND_AWAITING_LOCKIN")

    * **new_state** (string): New state (one of "OPENINGD", "CHANNELD_AWAITING_LOCKIN", "CHANNELD_NORMAL", "CHANNELD_SHUTTING_DOWN", "CLOSINGD_SIGEXCHANGE", "CLOSINGD_COMPLETE", "AWAITING_UNILATERAL", "FUNDING_SPEND_SEEN", "ONCHAIN", "DUALOPEND_OPEN_INIT", "DUALOPEND_AWAITING_LOCKIN")
    * **cause** (string): What caused the change (one of "unknown", "local", "user", "remote", "protocol", "onchain")
    * **message** (string): Human-readable explanation
  - **status** (array of strings, optional):
    * Billboard log of significant changes
  - **in_payments_offered** (u64, optional): Number of incoming payment attempts
  - **in_offered_msat** (msat, optional): Total amount of incoming payment attempts
  - **in_payments_fulfilled** (u64, optional): Number of successful incoming payment attempts
  - **in_fulfilled_msat** (msat, optional): Total amount of successful incoming payment attempts
  - **out_payments_offered** (u64, optional): Number of outgoing payment attempts
  - **out_offered_msat** (msat, optional): Total amount of outgoing payment attempts
  - **out_payments_fulfilled** (u64, optional): Number of successful outgoing payment attempts
  - **out_fulfilled_msat** (msat, optional): Total amount of successful outgoing payment attempts
  - **htlcs** (array of objects, optional): current HTLCs in this channel:
    * **direction** (string): Whether it came from peer, or is going to peer (one of "in", "out")
    * **id** (u64): Unique ID for this htlc on this channel in this direction
    * **amount_msat** (msat): Amount send/received for this HTLC
    * **expiry** (u32): Block this HTLC expires at
    * **payment_hash** (hash): the hash of the payment_preimage which will prove payment (always 64 characters)
    * **local_trimmed** (boolean, optional): if this is too small to enforce onchain (always *true*)
    * **status** (string, optional): set if this HTLC is currently waiting on a hook (and shows what plugin)

    If **direction** is "out":

    * **state** (string): Status of the HTLC (one of "SENT_ADD_HTLC", "SENT_ADD_COMMIT", "RCVD_ADD_REVOCATION", "RCVD_ADD_ACK_COMMIT", "SENT_ADD_ACK_REVOCATION", "RCVD_REMOVE_HTLC", "RCVD_REMOVE_COMMIT", "SENT_REMOVE_REVOCATION", "SENT_REMOVE_ACK_COMMIT", "RCVD_REMOVE_ACK_REVOCATION")

    If **direction** is "in":

    * **state** (string): Status of the HTLC (one of "RCVD_ADD_HTLC", "RCVD_ADD_COMMIT", "SENT_ADD_REVOCATION", "SENT_ADD_ACK_COMMIT", "RCVD_ADD_ACK_REVOCATION", "SENT_REMOVE_HTLC", "SENT_REMOVE_COMMIT", "RCVD_REMOVE_REVOCATION", "RCVD_REMOVE_ACK_COMMIT", "SENT_REMOVE_ACK_REVOCATION")

If **close_to** is present:

  - **close_to_addr** (string, optional): The bitcoin address we will close to

  If **scratch_txid** is present:

    - **last_tx_fee_msat** (msat): fee attached to this the current tx

  If **short_channel_id** is present:

    - **direction** (u32): 0 if we're the lesser node_id, 1 if we're the greater

  If **inflight** is present:

    - **initial_feerate** (string): The feerate for the initial funding transaction in per-1000-weight, with "kpw" appended
    - **last_feerate** (string): The feerate for the latest funding transaction in per-1000-weight, with "kpw" appended
    - **next_feerate** (string): The minimum feerate for the next funding transaction in per-1000-weight, with "kpw" appended

- **log** (array of objects, optional): if *level* is specified, logs for this peer:
  - **type** (string) (one of "SKIPPED", "BROKEN", "UNUSUAL", "INFO", "DEBUG", "IO_IN", "IO_OUT")

  If **type** is "SKIPPED":

    - **num_skipped** (u32): number of deleted/omitted entries

  If **type** is "BROKEN", "UNUSUAL", "INFO" or "DEBUG":

    - **time** (string): UNIX timestamp with 9 decimal places
    - **source** (string): The particular logbook this was found in
    - **log** (string): The actual log message
    - **node_id** (pubkey): The peer this is associated with

  If **type** is "IO_IN" or "IO_OUT":

    - **time** (string): UNIX timestamp with 9 decimal places
    - **source** (string): The particular logbook this was found in
    - **log** (string): The actual log message
    - **node_id** (pubkey): The peer this is associated with
    - **data** (hex): The IO which occurred

If **connected** is *true*:

- **netaddr** (array of strings): A single entry array:
  - address, e.g. 1.2.3.4:1234
- **features** (hex): bitmap of BOLT #9 features from peer's INIT message
- **remote_addr** (string, optional): The public IPv4/6 address the peer sees us from, e.g. 1.2.3.4:1234

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 array of objects describing channels with 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.

The objects in the *channels* array will have at least these fields:

- *state*: Any of these strings:
  - `"OPENINGD"`: The channel funding protocol with the peer is ongoing and both sides are negotiating parameters.
  - `"CHANNELD_AWAITING_LOCKIN"`: The peer and you have agreed on channel parameters and are just waiting for the channel funding transaction to be confirmed deeply. Both you and the peer must acknowledge the channel funding transaction to be confirmed deeply before entering the next state.
  - `"CHANNELD_NORMAL"`: The channel can be used for normal payments.
  - `"CHANNELD_SHUTTING_DOWN"`: A mutual close was requested (by you or peer) and both of you are waiting for HTLCs in-flight to be either failed or succeeded. The channel can no longer be used for normal payments and forwarding. Mutual close will proceed only once all HTLCs in the channel have either been fulfilled or failed.
  - `"CLOSINGD_SIGEXCHANGE"`: You and the peer are negotiating the mutual close onchain fee.
  - `"CLOSINGD_COMPLETE"`: You and the peer have agreed on the mutual close onchain fee and are awaiting the mutual close getting confirmed deeply.
  - `"AWAITING_UNILATERAL"`: You initiated a unilateral close, and are now waiting for the peer-selected unilateral close timeout to complete.
  - `"FUNDING_SPEND_SEEN"`: You saw the funding transaction getting spent (usually the peer initiated a unilateral close) and will now determine what exactly happened (i.e. if it was a theft attempt).
  - `"ONCHAIN"`: You saw the funding transaction getting spent and now know what happened (i.e. if it was a proper unilateral close by the peer, or a theft attempt).
  - `"CLOSED"`: The channel closure has been confirmed deeply. The channel will eventually be removed from this array.
- *state_changes*: An array of objects describing prior state change events.
- *opener*: A string `"local"` or `"remote`" describing which side opened this channel.
- *closer*: A string `"local"` or `"remote`" describing which side closed this channel or `null` if the channel is not (being) closed yet.
- *status*: An array of strings containing the most important log messages relevant to this channel. Also known as the "billboard".
- *owner*: A string describing which particular sub-daemon of `lightningd` currently is responsible for this channel. One of: `"lightning_openingd"`, `"lightning_channeld"`, `"lightning_closingd"`, `"lightning_onchaind"`.
- *to_us_msat*: A string describing how much of the funds is owned by us; a number followed by a string unit.
- *total_msat*: A string describing the total capacity of the channel; a number followed by a string unit.
- *fee_base_msat*: The fixed routing fee we charge for forwards going out over this channel, regardless of payment size.
- *fee_proportional_millionths*: The proportional routing fees in ppm (parts- per-millionths) we charge for forwards going out over this channel.
- *features*: An array of feature names supported by this channel.

These fields may exist if the channel has gotten beyond the `"OPENINGD"` state, or in various circumstances:

- *short_channel_id*: A string of the short channel ID for the channel; Format is `"BBBBxTTTxOOO"`, where `"BBBB"` is the numeric block height at which the funding transaction was confirmed, `"TTT"` is the numeric funding transaction index within that block, and `"OOO"` is the numeric output index of the transaction output that actually anchors this channel.
- *direction*: The channel-direction we own, as per BOLT #7. We own channel-direction 0 if our node ID is "less than" the peer node ID in a lexicographical ordering of our node IDs, otherwise we own channel-direction 1. Our `channel_update` will use this *direction*.
- *channel_id*: The full channel ID of the channel; the funding transaction ID XORed with the output number.
- *funding_txid*: The funding transaction ID of the channel.
- *close_to*: The raw `scriptPubKey` that was indicated in the starting **fundchannel_start** command and accepted by the peer. If the `scriptPubKey` encodes a standardized address, an additional *close_to_addr* field will be present with the standardized address.
- *private*: A boolean, true if the channel is unpublished, false if the channel is published.
- *funding_msat*: An object, whose field names are the node IDs involved in the channel, and whose values are strings (numbers with a unit suffix) indicating how much that node originally contributed in opening the channel.
- *min_to_us_msat*: A string describing the historic point at which we owned the least amount of funds in this channel; a number followed by a string unit. If the peer were to succesfully steal from us, this is the amount we would still retain.
- *max_to_us_msat*: A string describing the historic point at which we owned the most amount of funds in this channel; a number followed by a string unit. If we were to successfully steal from the peer, this is the amount we could potentially get.
- *dust_limit_msat*: A string describing an amount; if an HTLC or the amount wholly-owned by one node is at or below this amount, it will be considered "dusty" and will not appear in a close transaction, and will be donated to miners as fee; a number followed by a string unit.
- *max_total_htlc_in_msat*: A string describing an amount; the sum of all HTLCs in the channel cannot exceed this amount; a number followed by a string unit.
- *their_reserve_msat*: A string describing the minimum amount that the peer must keep in the channel when it attempts to send out; if it has less than this in the channel, it cannot send to us on that channel; a number followed by a string unit. We impose this on them, default is 1% of the total channel capacity.
- *our_reserve_msat*: A string describing the minimum amount that you must keep in the channel when you attempt to send out; if you have less than this in the channel, you cannot send out via this channel; a number followed by a string unit. The peer imposes this on us, default is 1% of the total channel capacity.
- *spendable_msat* and *receivable_msat*: A string describing an ***estimate*** of how much we can send or receive over this channel in a single payment (or payment-part for multi-part payments); a number followed by a string unit. This is an ***estimate***, which can be wrong because adding HTLCs requires an increase in fees paid to onchain miners, and onchain fees change dynamically according to onchain activity. For a sufficiently-large channel, this can be limited by the rules imposed under certain blockchains; for example, individual Bitcoin mainnet payment-parts cannot exceed 42.94967295 mBTC.
- *minimum_htlc_in_msat*: A string describing the minimum amount that an HTLC must have before we accept it.
- *their_to_self_delay*: The number of blocks that the peer must wait to claim their funds, if they close unilaterally.
- *our_to_self_delay*: The number of blocks that you must wait to claim your funds, if you close unilaterally.
- *max_accepted_htlcs*: The maximum number of HTLCs you will accept on this channel.
- *in_payments_offered*: The number of incoming HTLCs offered over this channel.

- *in_offered_msat*: A string describing the total amount of all incoming HTLCs offered over this channel; a number followed by a string unit.
- *in_payments_fulfilled*: The number of incoming HTLCs offered *and successfully claimed* over this channel.
- *in_fulfilled_msat*: A string describing the total amount of all incoming HTLCs offered *and successfully claimed* over this channel; a number followed by a string unit.
- *out_payments_offered*: The number of outgoing HTLCs offered over this channel.
- *out_offered_msat*: A string describing the total amount of all outgoing HTLCs offered over this channel; a number followed by a string unit.
- *out_payments_fulfilled*: The number of outgoing HTLCs offered *and successfully claimed* over this channel.
- *out_fulfilled_msat*: A string describing the total amount of all outgoing HTLCs offered *and successfully claimed* over this channel; a number followed by a string unit.
- *scratch_txid*: The txid of the latest transaction (what we would sign and send to chain if the channel were to fail now).
- *last_tx_fee*: The fee on that latest transaction.
- *feerate*: An object containing the latest feerate as both *perkw* and *perkb*.
- *htlcs*: An array of objects describing the HTLCs currently in-flight in the channel.

Objects in the *htlcs* array will contain these fields:

- *direction*: Either the string `"out"` or `"in"`, whether it is an outgoing or incoming HTLC.
- *id*: A numeric ID uniquely identifying this HTLC.
- *amount_msat*: The value of the HTLC.
- *expiry*: The blockheight at which the HTLC will be forced to return to its offerer: an `"in"` HTLC will be returned to the peer, an `"out"` HTLC will be returned to you. **NOTE** If the *expiry* of any outgoing HTLC will arrive in the next block, `lightningd`(8) will automatically unilaterally close the channel in order to enforce the timeout onchain.
- *payment_hash*: The payment hash, whose preimage must be revealed to successfully claim this HTLC.
- *state*: A string describing whether the HTLC has been communicated to or from the peer, whether it has been signed in a new commitment, whether the previous commitment (that does not contain it) has been revoked, as well as when the HTLC is fulfilled or failed offchain.
- *local_trimmed*: A boolean, existing and `true` if the HTLC is not actually instantiated as an output (i.e. "trimmed") on the commitment transaction (and will not be instantiated on a unilateral close). Generally true if the HTLC is below the *dust_limit_msat* for the channel.

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.

64.4 AUTHOR

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

64.5 SEE ALSO

lightning-connect(7), lightning-fundchannel_start(7), lightning-setchannel(7)

64.6 RESOURCES

Main web site: https://github.com/ElementsProject/lightning Lightning RFC site (BOLT #9): https://github.com/lightning/bolts/blob/master/09-features.md

CHAPTER 65

lightning-listsendpays – Low-level command for querying sendpay status

65.1 SYNOPSIS

listsendpays [*bolt11*] [*payment_hash*] [*status*]

65.2 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. It is possible filter the payments also by *status*.

Note that in future there may be more than one concurrent *sendpay* command per *pay*, so this command should be used with caution.

65.3 RETURN VALUE

Note that the returned array is ordered by increasing *id*.

On success, an object containing **payments** is returned. It is an array of objects, where each object contains:

- **id** (u64): unique ID for this payment attempt
- **groupid** (u64): Grouping key to disambiguate multiple attempts to pay an invoice or the same payment_hash
- **payment_hash** (hash): the hash of the *payment_preimage* which will prove payment (always 64 characters)
- **status** (string): status of the payment (one of "pending", "failed", "complete")
- **created_at** (u64): the UNIX timestamp showing when this payment was initiated
- **amount_sent_msat** (msat): The amount sent
- **amount_msat** (msat, optional): The amount delivered to destination (if known)

- **destination** (pubkey, optional): the final destination of the payment if known
- **label** (string, optional): the label, if given to sendpay
- **bolt11** (string, optional): the bolt11 string (if pay supplied one)
- **description** (string, optional): the description matching the bolt11 description hash (if pay supplied one)
- **bolt12** (string, optional): the bolt12 string (if supplied for pay: **experimental-offers** only).

If **status** is "complete":

- **payment_preimage** (secret): the proof of payment: SHA256 of this **payment_hash** (always 64 characters)

If **status** is "failed":

- **erroronion** (hex, optional): the onion message returned

65.4 AUTHOR

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

65.5 SEE ALSO

lightning-listpays(7), lightning-sendpay(7), lightning-listinvoice(7).

65.6 RESOURCES

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

CHAPTER 66

lightning-listtransactions – Command to get the list of transactions that was stored in the wallet.

66.1 SYNOPSIS

listtransactions

66.2 DESCRIPTION

The **listtransactions** command returns transactions tracked in the wallet. This includes deposits, withdrawals and transactions related to channels. A transaction may have multiple types, e.g., a transaction may both be a close and a deposit if it closes the channel and returns funds to the wallet.

66.3 EXAMPLE JSON REQUEST

```
{
  "id": 82,
  "method": "listtransactions",
  "params": {}
}
```

66.4 RETURN VALUE

On success, an object containing **transactions** is returned. It is an array of objects, where each object contains:

- **hash** (txid): the transaction id
- **rawtx** (hex): the raw transaction

- **blockheight** (u32): the block height of this tx
- **txindex** (u32): the transaction number within the block
- **locktime** (u32): The nLocktime for this tx
- **version** (u32): The nVersion for this tx
- **inputs** (array of objects): Each input, in order:
  - **txid** (txid): the transaction id spent
  - **index** (u32): the output spent
  - **sequence** (u32): the nSequence value
  - **type** (string, optional): the purpose of this input (*EXPERIMENTAL_FEATURES* only) (one of "theirs", "deposit", "withdraw", "channel_funding", "channel_mutual_close", "channel_unilateral_close", "channel_sweep", "channel_htlc_success", "channel_htlc_timeout", "channel_penalty", "channel_unilateral_cheat")
  - **channel** (short_channel_id, optional): the channel this input is associated with (*EXPERIMENTAL_FEATURES* only)
- **outputs** (array of objects): Each output, in order:
  - **index** (u32): the 0-based output number
  - **amount_msat** (msat): the amount of the output
  - **scriptPubKey** (hex): the scriptPubKey
  - **type** (string, optional): the purpose of this output (*EXPERIMENTAL_FEATURES* only) (one of "theirs", "deposit", "withdraw", "channel_funding", "channel_mutual_close", "channel_unilateral_close", "channel_sweep", "channel_htlc_success", "channel_htlc_timeout", "channel_penalty", "channel_unilateral_cheat")
  - **channel** (short_channel_id, optional): the channel this output is associated with (*EXPERIMENTAL_FEATURES* only)
- **type** (array of strings, optional):
  - Reason we care about this transaction (*EXPERIMENTAL_FEATURES* only) (one of "theirs", "deposit", "withdraw", "channel_funding", "channel_mutual_close", "channel_unilateral_close", "channel_sweep", "channel_htlc_success", "channel_htlc_timeout", "channel_penalty", "channel_unilateral_cheat")
- **channel** (short_channel_id, optional): the channel this transaction is associated with (*EXPERIMENTAL_FEATURES* only)

On failure, one of the following error codes may be returned:

- -32602: Error in given parameters.

66.5 EXAMPLE JSON RESPONSE

```
{
   "transactions": [
      {
         "hash": "05985072bbe20747325e69a159fe08176cc1bbc96d25e8848edad2dddc1165d0",
         "rawtx":
↪"02000000027032912651fc25a3e0893acd5f9640598707e2dfef92143bb5a4020e335442800100000017160014a5f48b9a
↪",
```

(continues on next page)

(continued from previous page)

```
         "blockheight": 0,
         "txindex": 0,
         "locktime": 0,
         "version": 2,
         "inputs": [
            {
               "txid":
↪"804254330e02a4b53b1492efdfe207875940965fcd3a89e0a325fc5126913270",
               "index": 1,
               "sequence": 4294967295
            },
            {
               "txid":
↪"4ee2eafa9a9b9e900aad69b7d3e4d6a5851556b0680ac1caeb3886f7b4a429d2",
               "index": 0,
               "sequence": 4294967295
            }
         ],
         "outputs": [
            {
               "index": 0,
               "satoshis": "88721000msat",
               "scriptPubKey": "a9143a4dfd59e781f9c3018e7d0a9b7a26d58f8d22bf87"
            }
         ]
      }
   ]
}
```

66.6 AUTHOR

Vincenzo Palazzo <vincenzo.palazzo@protonmail.com> wrote the initial version of this man page, but many others did the hard work of actually implementing this rpc command.

66.7 SEE ALSO

lightning-newaddr(7), lightning-listfunds(7)

66.8 RESOURCES

Main web site: https://github.com/ElementsProject/lightning *comment*: # (SHA256STAMP:f7c39908eaa1a2561597c8f97658b873953daab0a68ed2e9b68e434a55d55efe)

CHAPTER 67

lightning-makesecret – Command for deriving pseudorandom key from HSM

67.1 SYNOPSIS

makesecret [*hex*] [*string*]

67.2 DESCRIPTION

The **makesecret** RPC command derives a secret key from the HSM_secret.

One of *hex* or *string* must be specified: *hex* can be any hex data, *string* is a UTF-8 string interpreted literally.

67.3 RETURN VALUE

On success, an object is returned, containing:

- **secret** (secret): the pseudorandom key derived from HSM_secret (always 64 characters)

The following error codes may occur:

- -1: Catchall nonspecific error.

67.4 AUTHOR

Aditya <aditya.sharma20111@gmail.com> is mainly responsible.

67.5 RESOURCES

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

CHAPTER 68

lightning-multifundchannel – Command for establishing many lightning channels

68.1 SYNOPSIS

multifundchannel *destinations* [*feerate*] [*minconf*] [*utxos*] [*minchannels*] [*commitment_feerate*]

68.2 DESCRIPTION

The **multifundchannel** RPC command opens multiple payment channels with nodes by committing a single funding transaction to the blockchain that is shared by all channels.

If not already connected, **multifundchannel** will automatically attempt to connect; you may provide a *@host:port* hint appended to the node ID so that Core Lightning can learn how to connect to the node; see lightning-connect(7).

Once the transaction is confirmed, normal channel operations may begin. Readiness is indicated by **listpeers** reporting a *state* of `CHANNELD_NORMAL` for the channel.

destinations is an array of objects, with the fields:

- *id* is the node ID, with an optional *@host:port* appended to it in a manner understood by **connect**; see lightning-connect(7). Each entry in the *destinations* array must have a unique node *id*.
- *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 16,777,215 satoshi if more is available and large channels were not negotiated with the peer). 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 546 satoshi as of this writing, nor more than 16,777,215 satoshi (unless large channels were negotiated with the peer).
- *announce* is an optional flag that indicates whether to announce the channel with this, default `true`. If set to `false`, the channel is unpublished.
- *push_msat* is the amount of millisatoshis to outright give to the node. This is a gift to the peer, and you do not get a proof-of-payment out of this.

- *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.
- *request_amt* is the amount of liquidity you'd like to lease from peer. If peer supports `option_will_fund`, indicates to them to include this much liquidity into the channel. Must also pass in *compact_lease*.
- *compact_lease* is a compact represenation of the peer's expected channel lease terms. If the peer's terms don't match this set, we will fail to open the channel to this destination.

There must be at least one entry in *destinations*; it cannot be an empty array.

feerate is an optional feerate used for the opening transaction and, if *commitment_feerate* is not set, as the 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*.

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".

minchannels, if specified, will re-attempt funding as long as at least this many peers remain (must not be zero). The **multifundchannel** command will only fail if too many peers fail the funding process.

commitment_feerate is the initial feerate for commitment and HTLC transactions. See *feerate* for valid values.

68.3 RETURN VALUE

This command opens multiple channels with a single large transaction, thus only one transaction is returned.

If *minchannels* was specified and is less than the number of destinations, then it is possible that one or more of the destinations do not have a channel even if **multifundchannel** succeeded.

On success, an object is returned, containing:

- **tx** (hex): The raw transaction which funded the channel
- **txid** (txid): The txid of the transaction which funded the channel
- **channel_ids** (array of objects):
  - **id** (pubkey): The peer we opened the channel with
  - **outnum** (u32): The 0-based output index showing which output funded the channel
  - **channel_id** (hex): The channel_id of the resulting channel (always 64 characters)
  - **close_to** (hex, optional): The raw scriptPubkey which mutual close will go to; only present if *close_to* parameter was specified and peer supports `option_upfront_shutdown_script`
- **failed** (array of objects, optional): any peers we failed to open with (if *minchannels* was specified less than the number of destinations):
  - **id** (pubkey): The peer we failed to open the channel with
  - **method** (string): What stage we failed at (one of "connect", "openchannel_init", "fundchannel_start", "fundchannel_complete")
  - **error** (object):
    - **code** (integer): JSON error code from failing stage

  - ∗ **message** (string): Message from stage
  - ∗ **data**: Additional error data

On failure, none of the channels are created.

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.). See lightning-fundchannel_start(7) and lightning-fundchannel_complete(7).

There may be rare edge cases where a communications failure later in the channel funding process will cancel the funding locally, but the peer thinks the channel is already waiting for funding lockin. In that case, the next time we connect to the peer, our node will tell the peer to forget the channel, but some nodes (in particular, Core Lightning nodes) will disconnect when our node tells them to forget the channel. If you immediately **multifundchannel** with that peer, it could trigger this connect-forget-disconnect behavior, causing the second **multifundchannel** to fail as well due to disconnection. Doing a **connect** with the peers separately, and waiting for a few seconds, should help clear this hurdle; running **multifundchannel** a third time would also clear this.

68.4 AUTHOR

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

68.5 SEE ALSO

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

68.6 RESOURCES

Main web site: https://github.com/ElementsProject/lightning *comment*: # (SHA256STAMP:a507d57bbf36455924497c8354f41e225bc16f63f12fe01b4f7c4af37f0c6960)

CHAPTER 69

lightning-multiwithdraw – Command for withdrawing to multiple addresses

69.1 SYNOPSIS

multiwithdraw *outputs* [*feerate*] [*minconf*] [*utxos*]

69.2 DESCRIPTION

The **multiwithdraw** RPC command sends funds from Core Lightning's internal wallet to the addresses specified in *outputs*, which is an array containing objects of the form `{address: amount}`. The `amount` may be the string *"all"*, indicating that all onchain funds be sent to the specified address. 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.

69.3 RETURN VALUE

On success, an object is returned, containing:

- **tx** (hex): The raw transaction which was sent
- **txid** (txid): The txid of the **tx**

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.

69.4 AUTHOR

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

69.5 SEE ALSO

lightning-listfunds(7), lightning-fundchannel(7), lightning-newaddr(7), lightning-txprepare(7), lightning-withdraw(7).

69.6 RESOURCES

Main web site: https://github.com/ElementsProject/lightning *comment*: # (SHA256STAMP:6c0054088c17481dedbedb6a5ed4be7f09ce8783780707432907508ebf4bbd7a)

CHAPTER 70

lightning-newaddr – Command for generating a new address to be used by Core Lightning

70.1 SYNOPSIS

newaddr [*addresstype*]

70.2 DESCRIPTION

The **newaddr** RPC command generates a new address which can subsequently be used to fund channels managed by the Core 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 no *addresstype* is specified the address generated is a *bech32* address.

To send an on-chain payment *from* the Core Lightning node wallet, use `withdraw`.

70.3 RETURN VALUE

On success, an object is returned, containing:

- **bech32** (string, optional): The bech32 (native segwit) address
- **p2sh-segwit** (string, optional): The p2sh-wrapped address

70.4 ERRORS

If an unrecognized address type is requested an error message will be returned.

70.5 AUTHOR

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

70.6 SEE ALSO

lightning-listfunds(7), lightning-fundchannel(7), lightning-withdraw(7), lightning-listtransactions(7)

70.7 RESOURCES

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

CHAPTER 71

lightning-notifications – Command to set up notifications.

71.1 SYNOPSIS

notifications *enable*

71.2 DESCRIPTION

The **notifications** the RPC command enabled notifications for this JSON-RPC connection. By default (and for backwards-compatibility) notifications are disabled.

Various commands, especially complex and slow ones, offer notifications which indicate their progress.

- *enable*: *true* to enable notifications, *false* to disable them.

71.3 EXAMPLE JSON REQUEST

```
{
  "id": 82,
  "method": "notifications",
  "params": {
    "enable": true
  }
}
```

71.4 NOTIFICATIONS

Notifications are JSON-RPC objects without an *id* field. *lightningd* sends notifications (once enabled with this *notifications* command) with a *params id* field indicating which command the notification refers to.

Implementations should ignore notifications without an *id* parameter, or unknown *method*.

Common *method*s include:

- *message*: param *message*: a descriptional string indicating something which occurred relating to the command. Param *level* indicates the level, as per lightning-getlog(7): *info* and *debug* are typical.
- *progress*: param *num* and *total*, where *num* starts at 0 and is always less than *total*. Optional param *stage* with fields *num* and *total*, indicating what stage we are progressing through.

71.5 RETURN VALUE

On success, an empty object is returned.

On success, if *enable* was *true*, notifications will be forwarded from then on.

On failure, one of the following error codes may be returned:

- -32602: Error in given parameters.

71.6 EXAMPLE NOTIFICATIONS

```
{
   "method": "message",
   "params": {
       "id": 83,
       "message": "This is a test message",
       "level": "DEBUG"
   }
}
```

```
{
   "method": "progress",
   "params": {
       "id": 83,
       "num": 0,
       "total": 30
       "stage": {
           "num": 0,
           "total": 2
       }
   }
}
```

71.7 AUTHOR

Rusty Russell <rusty@blockstream.com> wrote the initial version of this man page.

71.8 RESOURCES

Main web site: https://github.com/ElementsProject/lightning *comment*: # (SHA256STAMP:326e5801f65998e13e909d8b682e9fbc9824f3a43aa7da1d76b871882e52f293)

CHAPTER 72

lightning-offer – Command for accepting payments

72.1 SYNOPSIS

(WARNING: experimental-offers only)

offer *amount description* [*issuer*] [*label*] [*quantity_max*] [*absolute_expiry*] [*recurrence*] [*recurrence_base*] [*recurrence_paywindow*] [*recurrence_limit*] [*single_use*]

72.2 DESCRIPTION

The **offer** RPC command creates an offer (or returns an existing one), which is a precursor to creating one or more invoices. It automatically enables the processing of an incoming invoice_request, and issuing of invoices.

Note that it creates two variants of the offer: a signed and an unsigned one (which is smaller). Wallets should accept both: the current specification allows either.

The *amount* parameter can be the string "any", which creates an offer that can be paid with any amount (e.g. a donation). Otherwise it can be a positive value 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*.

amount can also have an ISO 4217 postfix (i.e. USD), in which case currency conversion will need to be done for the invoice itself. A plugin is needed which provides the "currencyconvert" API for this currency, otherwise the offer creation will fail.

The *description* is a short description of purpose of the offer, e.g. *coffee*. This value is encoded into the resulting offer and is viewable by anyone you expose this offer to. It must be UTF-8, and cannot use *\u* JSON escape codes.

The *issuer* is another (optional) field exposed in the offer, and reflects who is issuing this offer (i.e. you) if appropriate.

The *label* field is an internal-use name for the offer, which can be any UTF-8 string.

The presence of *quantity_max* indicates that the invoice can specify more than one of the items up (and including) this maximum: 0 is a special value meaning "no maximuim". The *amount* for the invoice will need to be multiplied accordingly. This is encoded in the offer.

The *absolute_expiry* is optionally the time the offer is valid until, in seconds since the first day of 1970 UTC. If not set, the offer remains valid (though it can be deactivated by the issuer of course). This is encoded in the offer.

recurrence means that an invoice is expected at regular intervals. The argument is a positive number followed by one of "seconds", "minutes", "hours", "days", "weeks", "months" or "years" (variants without the trailing "s" are also permitted). This is encoded in the offer. The semantics of recurrence is fairly predictable, but fully documented in BOLT 12. e.g. "4weeks".

recurrence_base is an optional time in seconds since the first day of 1970 UTC, optionally with a "@" prefix. This indicates when the first period begins; without this, the recurrence periods start from the first invoice. The "@" prefix means that the invoice must start by paying the first period; otherwise it is permitted to start at any period. This is encoded in the offer. e.g. "@1609459200" indicates you must start paying on the 1st January 2021.

recurrence_paywindow is an optional argument of form '-time+time[%]'. The first time is the number of seconds before the start of a period in which an invoice and payment is valid, the second time is the number of seconds after the start of the period. For example *-604800+86400* means you can fetch an pay the invoice 4 weeks before the given period starts, and up to 1 day afterwards. The optional % indicates that the amount of the invoice will be scaled by the time remaining in the period. If this is not specified, the default is that payment is allowed during the current and previous periods. This is encoded in the offer.

recurrence_limit is an optional argument to indicate the maximum period which exists. eg. "12" means there are 13 periods, from 0 to 12 inclusive. This is encoded in the offer.

refund_for is the payment_preimage of a previous (paid) invoice. This implies *send_invoice* and *single_use*. This is encoded in the offer.

single_use (default false) indicates that the offer is only valid once; we may issue multiple invoices, but as soon as one is paid all other invoices will be expired (i.e. only one person can pay this offer).

72.3 RETURN VALUE

On success, an object is returned, containing:

- **offer_id** (hex): the id of this offer (merkle hash of non-signature fields) (always 64 characters)
- **active** (boolean): whether this can still be used (always *true*)
- **single_use** (boolean): whether this expires as soon as it's paid (reflects the *single_use* parameter)
- **bolt12** (string): the bolt12 encoding of the offer
- **used** (boolean): True if an associated invoice has been paid
- **created** (boolean): false if the offer already existed
- **label** (string, optional): the (optional) user-specified label

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

If the offer already existed, and is still active, that is returned; if it's not active then this call fails.

The following error codes may occur:

- -1: Catchall nonspecific error.
- 1000: Offer with this offer_id already exists (but is not active).

72.4 AUTHOR

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

72.5 SEE ALSO

lightning-offerout(7), lightning-listoffers(7), lightning-disableoffer(7).

72.6 RESOURCES

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

CHAPTER 73

lightning-offerout – Command for offering payments

73.1 SYNOPSIS

(WARNING: experimental-offers only)

offerout *amount description* [*issuer*] [*label*] [*absolute_expiry*] [*refund_for*]

73.2 DESCRIPTION

The **offerout** RPC command creates an offer, which is a request to send an invoice for us to pay (technically, this is referred to as a `send_invoice` offer to distinguish a normal lightningd-offer(7) offer). It automatically enables the accepting and payment of corresponding invoice message (we will only pay once, however!).

Note that it creates two variants of the offer: a signed and an unsigned one (which is smaller). Wallets should accept both: the current specification allows either.

The *amount* parameter can be the string "any", which creates an offer that can be paid with any amount (e.g. a donation). Otherwise it can be a positive value 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 *description* is a short description of purpose of the offer, e.g. *withdrawl from ATM*. This value is encoded into the resulting offer and is viewable by anyone you expose this offer to. It must be UTF-8, and cannot use *\u* JSON escape codes.

The *issuer* is another (optional) field exposed in the offer, and reflects who is issuing this offer (i.e. you) if appropriate.

The *label* field is an internal-use name for the offer, which can be any UTF-8 string.

The *absolute_expiry* is optionally the time the offer is valid until, in seconds since the first day of 1970 UTC. If not set, the offer remains valid (though it can be deactivated by the issuer of course). This is encoded in the offer.

refund_for is a previous (paid) invoice of ours. The payment_preimage of this is encoded in the offer, and redemption requires that the invoice we receive contains a valid signature using that previous `payer_key`.

73.3 RETURN VALUE

On success, an object is returned, containing:

- **offer_id** (hex): the id of this offer (merkle hash of non-signature fields) (always 64 characters)
- **active** (boolean): whether this will pay a matching incoming invoice (always *true*)
- **single_use** (boolean): whether this expires as soon as it's paid out (always *true*)
- **bolt12** (string): the bolt12 encoding of the offer
- **used** (boolean): True if an incoming invoice has been paid (always *false*)
- **created** (boolean): false if the offer already existed
- **label** (string, optional): the (optional) user-specified label

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

The following error codes may occur:

- -1: Catchall nonspecific error.
- 1000: Offer with this offer_id already exists.

73.4 NOTES

The specification allows quantity, recurrence and alternate currencies on offers which contain `send_invoice`, but these are not implemented here.

We could also allow multi-use offers, but usually you're only offering to send money once.

73.5 AUTHOR

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

73.6 SEE ALSO

lightning-sendinvoice(7), lightning-offer(7), lightning-listoffers(7), lightning-disableoffer(7).

73.7 RESOURCES

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

CHAPTER 74

lightning-openchannel_abort – Command to abort a channel to a peer

74.1 SYNOPSIS

openchannel_abort *channel_id*

74.2 DESCRIPTION

`openchannel_init` is a low level RPC command which initiates a channel open with a specified peer. It uses the openchannel protocol which allows for interactive transaction construction.

channel_id is id of this channel.

74.3 RETURN VALUE

On success, an object is returned, containing:

- **channel_id** (hex): the channel id of the aborted channel (always 64 characters)
- **channel_canceled** (boolean): whether this is completely canceled (there may be remaining in-flight transactions)
- **reason** (string): usually "Abort requested", but if it happened to fail at the same time it could be different

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.
- 311: Unknown channel id.
- 312: Channel in an invalid state

74.4 SEE ALSO

lightning-openchannel_init(7), lightning-openchannel_update(7), lightning-openchannel_signed(7), lightning-openchannel_bump(7), lightning-fundchannel_start(7), lightning-fundchannel_complete(7), lightning-fundchannel(7), lightning-fundpsbt(7), lightning-utxopsbt(7), lightning-multifundchannel(7)

74.5 AUTHOR

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

74.6 RESOURCES

Main web site: https://github.com/ElementsProject/lightning *comment*: # (SHA256STAMP:ed449af5b443c981faaff360cb2276816bbc7cd80f85fdb4403987c29d65baed)

CHAPTER 75

lightning-openchannel_bump – Command to initiate a channel RBF

75.1 SYNOPSIS

openchannel_bump *channel_id amount initalpsbt* [*funding_feerate*]

75.2 DESCRIPTION

`openchannel_bump` is a RPC command which initiates a channel RBF (Replace-By-Fee) for the specified channel. It uses the openchannel protocol which allows for interactive transaction construction.

id is the id of the channel to RBF.

amount is the satoshi value that we will contribute to the channel. This value will be *added* to the provided PSBT in the output which is encumbered by the 2-of-2 script for this channel.

initialpsbt is the funded, incomplete PSBT that specifies the UTXOs and change output for our channel contribution. It can be updated, see `openchannel_update`; *initialpsbt* must have at least one input. Must have the Non-Witness UTXO (PSBT_IN_NON_WITNESS_UTXO) set for every input. An error (code 309) will be returned if this requirement is not met.

funding_feerate is an optional field. Sets the feerate for the funding transaction. Defaults to 1/64th greater than the last feerate used for this channel.

Warning: bumping a leased channel will lose the lease.

75.3 RETURN VALUE

On success, an object is returned, containing:

- **channel_id** (hex): the channel id of the channel (always 64 characters)
- **psbt** (string): the (incomplete) PSBT of the RBF transaction

- **commitments_secured** (boolean): whether the *psbt* is complete (always *false*)
- **funding_serial** (u64): the serial_id of the funding output in the *psbt*

If the peer does not support `option_dual_fund`, this command will return an error.

If the channel is not in a state that is eligible for RBF, this command will return an error.

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 PSBT cannot afford the funding amount.
- 305: Peer is not connected.
- 309: PSBT missing required fields
- 311: Unknown channel id.
- 312: Channel in an invalid state

75.4 SEE ALSO

lightning-openchannel_init(7), lightning-openchannel_update(7), lightning-openchannel_signed(7), lightning-openchannel_abort(7), lightning-fundchannel_start(7), lightning-fundchannel_complete(7), lightning-fundchannel(7), lightning-fundpsbt(7), lightning-utxopsbt(7), lightning-multifundchannel(7)

75.5 AUTHOR

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

75.6 RESOURCES

Main web site: https://github.com/ElementsProject/lightning *comment*: # (SHA256STAMP:3cba5d1c16925322754eae979e956132e8b94e40da0dee6925037a8854d9b791)

CHAPTER 76

lightning-openchannel_init – Command to initiate a channel to a peer

76.1 SYNOPSIS

openchannel_init *id amount initalpsbt* [*commitment_feerate*] [*funding_feerate*] [*announce*] [*close_to*] [*request_amt*] [*compact_lease*]

76.2 DESCRIPTION

`openchannel_init` is a low level RPC command which initiates a channel open with a specified peer. It uses the openchannel protocol which allows for interactive transaction construction.

id is the node id of the remote peer.

amount is the satoshi value that we will contribute to the channel. This value will be *added* to the provided PSBT in the output which is encumbered by the 2-of-2 script for this channel.

initialpsbt is the funded, incomplete PSBT that specifies the UTXOs and change output for our channel contribution. It can be updated, see `openchannel_update`; *initialpsbt* must have at least one input. Must have the Non-Witness UTXO (PSBT_IN_NON_WITNESS_UTXO) set for every input. An error (code 309) will be returned if this requirement is not met.

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

funding_feerate is an optional field. Sets the feerate for the funding transaction. Defaults to 'opening' feerate.

announce is an optional field. Whether or not to announce this channel.

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

request_amt is an amount of liquidity you'd like to lease from the peer. If peer supports `option_will_fund`, indicates to them to include this much liquidity into the channel. Must also pass in *compact_lease*.

compact_lease is a compact represenation of the peer's expected channel lease terms. If the peer's terms don't match this set, we will fail to open the channel.

76.3 RETURN VALUE

On success, an object is returned, containing:

- **channel_id** (hex): the channel id of the channel (always 64 characters)
- **psbt** (string): the (incomplete) PSBT of the funding transaction
- **commitments_secured** (boolean): whether the *psbt* is complete (always *false*)
- **funding_serial** (u64): the serial_id of the funding output in the *psbt*

If the peer does not support `option_dual_fund`, this command will return an error.

If you sent a *request_amt* and the peer supports `option_will_fund` and is interested in leasing you liquidity in this channel, returns their updated channel fee max (*channel_fee_proportional_basis*, *channel_fee_base_msat*), updated rate card for the lease fee (*lease_fee_proportional_basis*, *lease_fee_base_sat*) and their on-chain weight *weight_charge*, which will be added to the lease fee at a rate of *funding_feerate* * *weight_charge* / 1000.

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 PSBT cannot afford the funding amount.
- 304: Still syncing with bitcoin network
- 305: Peer is not connected.
- 306: Unknown peer id.
- 309: PSBT missing required fields
- 310: v2 channel open protocol not supported by peer
- 312: Channel in an invalid state

76.4 SEE ALSO

lightning-openchannel_update(7), lightning-openchannel_signed(7), lightning-openchannel_abort(7), lightning-openchannel_bump(7), lightning-fundchannel_start(7), lightning-fundchannel_complete(7), lightning-fundchannel(7), lightning-fundpsbt(7), lightning-utxopsbt(7), lightning-multifundchannel(7)

76.5 AUTHOR

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

76.6 RESOURCES

Main web site: https://github.com/ElementsProject/lightning *comment*: # (SHA256STAMP:18421f03dece31aafe32cb1a9b520dd6b898e018cb187de6d666e391232fab4e)

CHAPTER 77

lightning-openchannel_signed – Command to conclude a channel open

77.1 SYNOPSIS

openchannel_signed *channel_id signed_psbt*

77.2 DESCRIPTION

`openchannel_signed` is a low level RPC command which concludes a channel open with the specified peer. It uses the v2 openchannel protocol, which allows for interactive transaction construction.

This command should be called after `openchannel_update` returns *commitments_secured* `true`.

This command will broadcast the finalized funding transaction, if we receive valid signatures from the peer.

channel_id is the id of the channel.

signed_psbt is the PSBT returned from `openchannel_update` (where *commitments_secured* was true) with partial signatures or finalized witness stacks included for every input that we contributed to the PSBT.

77.3 RETURN VALUE

On success, an object is returned, containing:

- **channel_id** (hex): the channel id of the channel (always 64 characters)
- **tx** (hex): the funding transaction
- **txid** (txid): The txid of the **tx**

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.

- 303: Funding transaction broadcast failed.
- 305: Peer is not connected.
- 309: PSBT missing required fields.
- 311: Unknown channel id.
- 312: Channel in an invalid state

77.4 SEE ALSO

lightning-openchannel_init(7), lightning-openchannel_update(7), lightning-openchannel_abort(7), lightning-openchannel_bump(7), lightning-fundchannel_start(7), lightning-fundchannel_complete(7), lightning-fundchannel(7), lightning-fundpsbt(7), lightning-utxopsbt(7), lightning-multifundchannel(7)

77.5 AUTHOR

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

77.6 RESOURCES

Main web site: https://github.com/ElementsProject/lightning *comment*: # (SHA256STAMP:41e2a4aed1aaac01675f99e91326197afa370a05e32b2ef20cbbb8247de57289)

CHAPTER 78

lightning-openchannel_update – Command to update a collab channel open

78.1 SYNOPSIS

openchannel_update *channel_id psbt*

78.2 DESCRIPTION

`openchannel_update` is a low level RPC command which continues an open channel, as specified by *channel_id*. An updated *psbt* is passed in; any changes from the PSBT last returned (either from `openchannel_init` or a previous call to `openchannel_update`) will be communicated to the peer.

Must be called after `openchannel_init` and before `openchannel_signed`.

Must be called until *commitments_secured* is returned as true, at which point `openchannel_signed` should be called with a signed version of the PSBT returned by the last call to `openchannel_update`.

channel_id is the id of the channel.

psbt is the updated PSBT to be sent to the peer. May be identical to the PSBT last returned by either `openchannel_init` or `openchannel_update`.

78.3 RETURN VALUE

On success, an object is returned, containing:

- **channel_id** (hex): the channel id of the channel (always 64 characters)
- **psbt** (string): the PSBT of the funding transaction
- **commitments_secured** (boolean): whether the *psbt* is complete (if true, sign *psbt* and call `openchannel_signed` to complete the channel open)

- **funding_outnum** (u32): The index of the funding output in the psbt
- **close_to** (hex, optional): scriptPubkey which we have to close to if we mutual close

If *commitments_secured* is true, will also return:

- The derived *channel_id*.
- A *close_to* script, iff a `close_to` address was provided to `openchannel_init` and the peer supports `option_upfront_shutdownscript`.
- The *funding_outnum*, the index of the funding output for this channel in the funding transaction.
- -32602: If the given parameters are wrong.
- -1: Catchall nonspecific error.
- 305: Peer is not connected.
- 309: PSBT missing required fields
- 311: Unknown channel id.
- 312: Channel in an invalid state

78.4 SEE ALSO

lightning-openchannel_init(7), lightning-openchannel_signed(7), lightning-openchannel_bump(7), lightning-openchannel_abort(7), lightning-fundchannel_start(7), lightning-fundchannel_complete(7), lightning-fundchannel(7), lightning-fundpsbt(7), lightning-utxopsbt(7), lightning-multifundchannel(7)

78.5 AUTHOR

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

78.6 RESOURCES

Main web site: https://github.com/ElementsProject/lightning *comment*: # (SHA256STAMP:14632f65d4c44b34762d3fa7e0f5b823a519d3dc5fc7a2a69f677000efd937fb)

CHAPTER 79

lightning-parsefeerate – Command for parsing a feerate string to a feerate

79.1 SYNOPSIS

parsefeerate *feerate_str*

79.2 DESCRIPTION

The **parsefeerate** command returns the current feerate for any valid *feerate_str*. This is useful for finding the current feerate that a **fundpsbt** or **utxopsbt** command might use.

79.3 RETURN VALUE

On success, an object is returned, containing:

- **perkw** (u32, optional): Value of *feerate_str* in kilosipa

79.4 ERRORS

The **parsefeerate** command will error if the *feerate_str* format is not recognized.

- -32602: If the given parameters are wrong.

79.5 TRIVIA

In CLN we like to call the weight unit “sipa” in honor of Pieter Wuille, who uses the name “sipa” on IRC and elsewhere. Internally we call the *perkw* style as “feerate per kilosipa”.

79.6 RESOURCES

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

CHAPTER 80

lightning-pay – Command for sending a payment to a BOLT11 invoice

80.1 SYNOPSIS

pay *bolt11* [*msatoshi*] [*label*] [*riskfactor*] [*maxfeepercent*] [*retry_for*] [*maxdelay*] [*exemptfee*] [*localinvreqid*] [*exclude*] [*maxfee*] [*description*]

80.2 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*.

(Note: if **experimental-offers** is enabled, *bolt11* can actually be a bolt12 invoice, such as one received from lightningd-fetchinvoice(7)).

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).

`localinvreqid` is used by offers to link a payment attempt to a local `invoice_request` offer created by lightningd-invoicerequest(7). This ensures that we only make a single payment for an offer, and that the offer is marked `used` once paid.

maxfee overrides both *maxfeepercent* and *exemptfee* defaults (and if you specify *maxfee* you cannot specify either of those), and creates an absolute limit on what fee we will pay. This allows you to implement your own heuristics rather than the primitive ones used here.

description is only required for bolt11 invoices which do not contain a description themselves, but contain a description hash. *description* is then checked against the hash inside the invoice before it will be paid.

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.

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.

When using *lightning-cli*, you may skip optional parameters by using *null*. Alternatively, use **-k** option to provide parameters by name.

80.3 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*.

80.4 RETURN VALUE

On success, an object is returned, containing:

- **payment_preimage** (secret): the proof of payment: SHA256 of this **payment_hash** (always 64 characters)
- **payment_hash** (hash): the hash of the *payment_preimage* which will prove payment (always 64 characters)
- **created_at** (number): the UNIX timestamp showing when this payment was initiated
- **parts** (u32): how many attempts this took
- **amount_msat** (msat): Amount the recipient received
- **amount_sent_msat** (msat): Total amount we sent (including fees)
- **status** (string): status of payment (one of "complete", "pending", "failed")
- **destination** (pubkey, optional): the final destination of the payment

The following warnings may also be returned:

- **warning_partial_completion**: Not all parts of a multi-part payment have completed

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.

80.5 AUTHOR

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

80.6 SEE ALSO

lightning-listpays(7), lightning-decodepay(7), lightning-listinvoice(7), lightning-delinvoice(7), lightning-getroute(7), lightning-invoice(7).

80.7 RESOURCES

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

CHAPTER 81

lightning-ping – Command to check if a node is up.

81.1 SYNOPSIS

ping *id* [*len*] [*pongbytes*]

81.2 DESCRIPTION

The **ping** command checks if the node with *id* is ready to talk. It currently only works for peers we have a channel with.

It accepts the following parameters:

- *id*: A string that represents the node id;
- *len*: A integer that represents the length of the ping (default 128);
- *pongbytes*: An integer that represents the length of the reply (default 128). A value of 65532 to 65535 means "don't reply".

81.3 EXAMPLE JSON REQUEST

```
{
  "id": 82,
  "method": "ping",
  "params": {
    "len": 128,
    "pongbytes": 128
  }
}
```

81.4 RETURN VALUE

On success, an object is returned, containing:

- **totlen** (u16): the answer length of the reply message (including header: 0 means no reply expected)

On failure, one of the following error codes may be returned:

- -32602: Error in given parameters or we're already waiting for a ping response from peer.

81.5 EXAMPLE JSON RESPONSE

```
{
   "totlen": 132
}
```

81.6 AUTHOR

Vincenzo Palazzo <vincenzo.palazzo@protonmail.com> wrote the initial version of this man page, but many others did the hard work of actually implementing this rpc command.

81.7 SEE ALSO

lightning-connect(7)

81.8 RESOURCES

Main web site: https://github.com/ElementsProject/lightning *comment*: # (SHA256STAMP:f33aa4d93ca623ff7cd5e4062e0533f617b00372797f8ee0d2498479d2fe08a9)

CHAPTER 82

lightning-plugin – Manage plugins with RPC

82.1 SYNOPSIS

plugin *subcommand* [plugin|directory] [*options*] . . .

82.2 DESCRIPTION

The **plugin** RPC command command can be used to control dynamic plugins, i.e. plugins that declared themself "dynamic" (in getmanifest).

subcommand can be **start**, **stop**, **startdir**, **rescan** or **list** and determines what action is taken

plugin is the *path* or *name* of a plugin executable to start or stop

directory is the *path* of a directory containing plugins

options are optional *keyword=value* options passed to plugin, can be repeated

subcommand **start** takes a *path* to an executable as argument and starts it as plugin. *path* may be an absolute path or a path relative to the plugins directory (default *~/.lightning/plugins*). If the plugin is already running and the executable (checksum) has changed, the plugin is killed and restarted except if its an important (or builtin) plugin. If the plugin doesn't complete the "getmanifest" and "init" handshakes within 60 seconds, the command will timeout and kill the plugin. Additional *options* may be passed to the plugin, but requires all parameters to be passed as keyword=value pairs, for example: `lightning-cli -k plugin subcommand=start plugin=helloworld.py greeting='A crazy'` (using the `-k|--keyword` option is recommended)

subcommand **stop** takes a plugin executable *path* or *name* as argument and stops the plugin. If the plugin subscribed to "shutdown", it may take up to 30 seconds before this command returns. If the plugin is important and dynamic, this will shutdown `lightningd`.

subcommand **startdir** starts all executables it can find in *directory* (excl. subdirectories) as plugins. Checksum and timeout behavior as in **start** applies.

subcommand **rescan** starts all plugins in the default plugins directory (default *~/.lightning/plugins*) that are not already running. Checksum and timeout behavior as in **start** applies.

subcommand **list** lists all running plugins (incl. non-dynamic)

82.3 RETURN VALUE

On success, an object is returned, containing:

- **command** (string): the subcommand this is responding to (one of "start", "stop", "rescan", "startdir", "list")

If **command** is "start", "startdir", "rescan" or "list":

- **plugins** (array of objects):
  - **name** (string): full pathname of the plugin
  - **active** (boolean): status; plugin completed init and is operational, plugins are configured asynchronously.
  - **dynamic** (boolean): plugin can be stopped or started without restarting lightningd

If **command** is "stop":

- **result** (string): A message saying it successfully stopped

On error, the reason why the action could not be taken upon the plugin is returned.

82.4 SEE ALSO

lightning-cli(1), lightning-listconfigs(1), *writing plugins*

82.5 AUTHOR

Antoine Poinsot <darosior@protonmail.com> is mainly responsible.

82.6 RESOURCES

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

CHAPTER 83

lightning-recoverchannel – Command for recovering channels bundeled in an array in the form of *Static Backup*

83.1 SYNOPSIS

recoverchannel *scb*

83.2 DESCRIPTION

The **recoverchannel** RPC command tries to force the peer (with whom you already had a channel) to close the channel and sweeps on-chain fund. This method is not spontaneous and depends on the peer, so use it in case of severe data loss.

The *scb* parameter is an array containing minimum required info to reconnect and sweep funds. You can get the scb for already stored channels by using the RPC command 'staticbackup'

83.3 RETURN VALUE

On success, an object is returned, containing:

- **stubs** (array of hexs):
  - Each item is the channel ID of the channel successfully inserted

83.4 AUTHOR

Aditya <aditya.sharma20111@gmail.com> is mainly responsible.

83.5 SEE ALSO

lightning-getsharedsecret(7)

83.6 RESOURCES

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

CHAPTER 84

lightning-reserveinputs – Construct a transaction and reserve the UTXOs it spends

84.1 SYNOPSIS

reserveinputs *psbt* [*exclusive*] [*reserve*]

84.2 DESCRIPTION

The **reserveinputs** RPC command places (or increases) reservations on any inputs specified in *psbt* which are known to lightningd. It will fail with an error if any of the inputs are known to be spent, and ignore inputs which are unknown.

Normally the command will fail (with no reservations made) if an input is already reserved. If *exclusive* is set to *False*, then existing reservations are simply extended, rather than causing failure.

By default, reservations are for the next 72 blocks (approximately 6 hours), but this can be changed by setting *reserve*.

84.3 RETURN VALUE

On success, an object containing **reservations** is returned. It is an array of objects, where each object contains:

- **txid** (txid): the transaction id
- **vout** (u32): the output number which was reserved
- **was_reserved** (boolean): whether the input was already reserved
- **reserved** (boolean): whether the input is now reserved (always *true*)
- **reserved_to_block** (u32): what blockheight the reservation will expire

On success, a *reservations* array is returned, with an entry for each input which was reserved:

- *txid* is the input transaction id.

- *vout* is the input index.
- *was_reserved* indicates whether the input was already reserved.
- *reserved* indicates that the input is now reserved (i.e. true).
- *reserved_to_block* indicates what blockheight the reservation will expire.

On failure, an error is reported and no UTXOs are reserved.

The following error codes may occur:

- -32602: Invalid parameter, such as specifying a spent/reserved input in *psbt*.

84.4 AUTHOR

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

84.5 SEE ALSO

lightning-unreserveinputs(7), lightning-signpsbt(7), lightning-sendpsbt(7)

84.6 RESOURCES

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

CHAPTER 85

lightning-sendcustommsg – Low-level interface to send protocol messages to peers

85.1 SYNOPSIS

sendcustommsg *node_id msg*

85.2 DESCRIPTION

The `sendcustommsg` RPC method allows the user to inject a custom message into the communication with the peer with the given `node_id`. This is intended as a low-level interface to implement custom protocol extensions on top, not for direct use by end-users.

The message must be a hex encoded well-formed message, including the 2-byte type prefix, but excluding the length prefix which will be added by the RPC method. The messages must not use even-numbered types, since these may require synchronous handling on the receiving side, and can cause the connection to be dropped. The message types may also not use one of the internally handled types, since that may cause issues with the internal state tracking of Core Lightning.

The node specified by `node_id` must be a peer, i.e., it must have a direct connection with the node receiving the RPC call, and the connection must be established. For a method to send arbitrary messages over multiple hops, including hops that do not understand the custom message, see the `createonion` and `sendonion` RPC methods. Messages can only be injected if the connection is handled by `openingd` or `channeld`. Messages cannot be injected when the peer is handled by `onchaind` or `closingd` since these do not have a connection, or are synchronous daemons that do not handle spontaneous messages.

On the reveiving end a plugin may implement the `custommsg` plugin hook and get notified about incoming messages.

85.3 RETURN VALUE

The method will validate the arguments and queue the message for delivery through the daemon that is currently handling the connection. Queuing provides best effort guarantees and the message may not be delivered if the connection is terminated while the message is queued. The RPC method will return as soon as the message is queued.

If any of the above limitations is not respected the method returns an explicit error message stating the issue.

On success, an object is returned, containing:

- **status** (string): Information about where message was queued

85.4 AUTHOR

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

85.5 SEE ALSO

lightning-createonion(7), lightning-sendonion(7)

85.6 RESOURCES

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

CHAPTER 86

lightning-sendinvoice – Command for send an invoice for an offer

86.1 SYNOPSIS

(WARNING: experimental-offers only)

sendinvoice *offer label* [*msatoshi*] [*timeout*] [*quantity*]

86.2 DESCRIPTION

The **sendinvoice** RPC command creates and sends an invoice to the issuer of an *offer* for it to pay: the offer must contain *send_invoice*; see lightning-fetchinvoice(7).

If **fetchinvoice-noconnect** is not specified in the configuation, it will connect to the destination in the (currently common!) case where it cannot find a route which supports `option_onion_messages`.

offer is the bolt12 offer string beginning with "lno1".

label is the unique label to use for this invoice.

msatoshi is optional: it is required if the *offer* does not specify an amount at all, or specifies it in a different currency. Otherwise you may set it (e.g. to provide a tip), and if not it defaults to the amount contained in the offer (multiplied by *quantity* if any).

timeout is how many seconds to wait for the offering node to pay the invoice or return an error, default 90 seconds. This will also be the timeout on the invoice that is sent.

quantity is optional: it is required if the *offer* specifies *quantity_min* or *quantity_max*, otherwise it is not allowed.

86.3 RETURN VALUE

On success, an object is returned, containing:

- **label** (string): unique label supplied at invoice creation

- **description** (string): description used in the invoice
- **payment_hash** (hex): the hash of the *payment_preimage* which will prove payment (always 64 characters)
- **status** (string): Whether it's paid, unpaid or unpayable (one of "unpaid", "paid", "expired")
- **expires_at** (u64): UNIX timestamp of when it will become / became unpayable
- **amount_msat** (msat, optional): the amount required to pay this invoice
- **bolt12** (string, optional): the BOLT12 string

If **status** is "paid":

- **pay_index** (u64): Unique incrementing index for this payment
- **amount_received_msat** (msat): the amount actually received (could be slightly greater than *amount_msat*, since clients may overpay)
- **paid_at** (u64): UNIX timestamp of when it was paid
- **payment_preimage** (hex): proof of payment (always 64 characters)

The following error codes may occur:

- -1: Catchall nonspecific error.
- 1002: Offer has expired.
- 1003: Cannot find a route to the node making the offer.
- 1004: The node making the offer returned an error message.
- 1005: We timed out waiting for the invoice to be paid

86.4 AUTHOR

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

86.5 SEE ALSO

lightning-fetchinvoice(7).

86.6 RESOURCES

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

CHAPTER 87

lightning-sendonion – Send a payment with a custom onion packet

87.1 SYNOPSIS

sendonion *onion first_hop payment_hash* [*label*] [*shared_secrets*] [*partid*] [*bolt11*] [*msatoshi*] [*destination*]

87.2 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 Core 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 Core 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 Core Lightning to use any available channel to `022d223620a359a47ff7f7ac447c85c46c923da53389221a0054c11c1e3ca31d59` to add an HTLC for 1002 millisatoshis and a delay of 21 blocks on top of the current blockheight:

```
{
  "id": "022d223620a359a47ff7f7ac447c85c46c923da53389221a0054c11c1e3ca31d59",
  "amount_msat": "1002msat",
  "delay": 21,
}
```

If the first element of *route* does not have "channel" set, a suitable channel (if any) will be chosen, otherwise that specific short-channel-id is used.

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. Core 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 Core 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 Core 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 Core 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.

The *partid* value, if provided and non-zero, allows for multiple parallel partial payments with the same *payment_hash*.

The *bolt11* parameter, if provided, will be returned in *waitsendpay* and *listsendpays* results.

The *destination* parameter, if provided, will be returned in **listpays** result.

The *msatoshi* parameter is used to annotate the payment, and is returned by *waitsendpay* and *listsendpays*.

87.3 RETURN VALUE

On success, an object is returned, containing:

- **id** (u64): unique ID for this payment attempt
- **payment_hash** (hash): the hash of the *payment_preimage* which will prove payment (always 64 characters)
- **status** (string): status of the payment (could be complete if already sent previously) (one of "pending", "complete")
- **created_at** (u64): the UNIX timestamp showing when this payment was initiated
- **amount_sent_msat** (msat): The amount sent
- **amount_msat** (msat, optional): The amount delivered to destination (if known)
- **destination** (pubkey, optional): the final destination of the payment if known
- **label** (string, optional): the label, if given to sendpay
- **bolt11** (string, optional): the bolt11 string (if supplied)
- **bolt12** (string, optional): the bolt12 string (if supplied: **experimental-offers** only).
- **partid** (u64, optional): the partid (if supplied) to sendonion/sendpay

If **status** is "complete":

- **payment_preimage** (secret): the proof of payment: SHA256 of this **payment_hash** (always 64 characters)

If **status** is “pending”:

- **message** (string, optional): Monitor status with listpays or waitsendpay

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.

87.4 AUTHOR

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

87.5 SEE ALSO

lightning-createonion(7), lightning-sendpay(7), lightning-listsendpays(7)

87.6 RESOURCES

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

CHAPTER 88

lightning-sendonionmessage – low-level command to send an onion message

88.1 SYNOPSIS

(WARNING: experimental-onion-messages only)

sendonionmessage *first_id blinding hops*

88.2 DESCRIPTION

The **sendonionmessage** RPC command can be used to send a message via the lightning network. These are currently used by *offers* to request and receive invoices.

hops is an array of json objects: *id* as a public key of the node, and *tlv* contains a hexidecimal TLV to include.

88.3 RETURN VALUE

On success, an empty object is returned.

88.4 AUTHOR

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

88.5 SEE ALSO

lightning-fetchinvoice(7), lightning-offer(7).

88.6 RESOURCES

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

CHAPTER 89

lightning-sendpay – Low-level command for sending a payment via a route

89.1 SYNOPSIS

sendpay *route payment_hash* [*label*] [*msatoshi*] [*bolt11*] [*payment_secret*] [*partid*] [*localinvreqid*] [*groupid*] [*payment_metadata*] [*description*]

89.2 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 must be provided if *partid* is non-zero, otherwise it must be equal to 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 *payment_secret* is the value that the final recipient requires to accept the payment, as defined by the `payment_data` field in BOLT 4 and the `s` field in the BOLT 11 invoice format. It is required if *partid* is non-zero.

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.

The *localinvreqid* value indicates that this payment is being made for a local invoice_request: this ensures that we only send a payment for a single-use invoice_request once.

groupid allows you to attach a number which appears in **listsendpays** so payments can be identified as part of a logical group. The *pay* plugin uses this to identify one attempt at a MPP payment, for example.

payment_metadata is placed in the final onion hop TLV.

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.

89.3 RETURN VALUE

On success, an object is returned, containing:

- **id** (u64): unique ID for this payment attempt
- **payment_hash** (hash): the hash of the *payment_preimage* which will prove payment (always 64 characters)
- **status** (string): status of the payment (could be complete if already sent previously) (one of "pending", "complete")
- **created_at** (u64): the UNIX timestamp showing when this payment was initiated
- **amount_sent_msat** (msat): The amount sent
- **groupid** (u64, optional): Grouping key to disambiguate multiple attempts to pay an invoice or the same payment_hash
- **amount_msat** (msat, optional): The amount delivered to destination (if known)
- **destination** (pubkey, optional): the final destination of the payment if known
- **completed_at** (u64, optional): the UNIX timestamp showing when this payment was completed
- **label** (string, optional): the *label*, if given to sendpay
- **partid** (u64, optional): the *partid*, if given to sendpay
- **bolt11** (string, optional): the bolt11 string (if supplied)
- **bolt12** (string, optional): the bolt12 string (if supplied: **experimental-offers** only).

If **status** is "complete":

- **payment_preimage** (secret): the proof of payment: SHA256 of this **payment_hash** (always 64 characters)

If **status** is "pending":

- **message** (string): Monitor status with listpays or waitsendpay

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.

- 212: *localinvreqid* refers to an invalid, or used, local invoice_request.

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. In addition *erring_direction* will indicate which direction of the channel caused the failure.
- *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.

89.4 AUTHOR

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

89.5 SEE ALSO

lightning-listinvoice(7), lightning-delinvoice(7), lightning-getroute(7), lightning-invoice(7), lightning-pay(7), lightning-waitsendpay(7).

89.6 RESOURCES

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

CHAPTER 90

lightning-sendpsbt – Command to finalize, extract and send a partially signed bitcoin transaction (PSBT).

90.1 SYNOPSIS

sendpsbt *psbt* [*reserve*]

90.2 DESCRIPTION

The **sendpsbt** is a low-level RPC command which sends a fully-signed PSBT.

- *psbt*: A string that represents psbt value.
- *reserve*: an optional number of blocks to increase reservation of any of our inputs by; default is 72.

90.3 EXAMPLE JSON REQUEST

```
{
  "id": 82,
  "method": "sendpsbt",
  "params": {
    "psbt": "some_psbt"
  }
}
```

90.4 RETURN VALUE

On success, an object is returned, containing:

- **tx** (hex): The raw transaction which was sent
- **txid** (txid): The txid of the **tx**

On failure, one of the following error codes may be returned:

- -32602: Error in given parameters or some error happened during the command process.

90.5 EXAMPLE JSON RESPONSE

```
{
    "txid": "05985072bbe20747325e69a159fe08176cc1bbc96d25e8848edad2dddc1165d0",
    "tx":
↪"02000000027032912651fc25a3e0893acd5f9640598707e2dfef92143bb5a4020e335442800100000017160014a5f48b9a
↪",
}
```

90.6 AUTHOR

Vincenzo Palazzo <vincenzo.palazzo@protonmail.com> wrote the initial version of this man page, but many others did the hard work of actually implementing this rpc command.

90.7 SEE ALSO

lightning-fundpsbt(7), lightning-signpsbt(7), lightning-listtransactions(7)

90.8 RESOURCES

Main web site: https://github.com/ElementsProject/lightning *comment*: # (SHA256STAMP:6c0054088c17481dedbedb6a5ed4be7f09ce878378070743290 7508ebf4bbd7a)

CHAPTER 91

lightning-setchannel – Command for configuring fees / htlc range advertized for a channel

91.1 SYNOPSIS

setchannel *id* [*feebase*] [*feeppm*] [*htlcmin*] [*htlcmax*] [*enforcedelay*]

91.2 DESCRIPTION

The **setchannel** RPC command sets channel specific routing fees, and `htlc_minimum_msat` or `htlc_maximum_msat` 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.

These changes (for a public channel) will be broadcast to the rest of the network (though many nodes limit the rate of such changes they will accept: we allow 2 a day, with a few extra occasionally).

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 updates are applied to all channels in states CHANNELD_NORMAL CHANNELD_AWAITING_LOCKIN or DUALOPEND_AWAITING_LOCKIN. If *id* is a peerid, all channels with the +peer in those states are changed.

feebase is an optional value in millisatoshi that is added as base fee to any routed payment: if omitted, it is unchanged. 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*.

feeppm 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 through the channel, an proportional fee of 1,000 satoshi is added, resulting in a 0.1% fee.

htlcmin is an optional value that limits how small an HTLC we will forward: if omitted, it is unchanged (the default is no lower limit). 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*. Note that the peer also enforces a minimum for the channel: setting it below that will simply set it to that value with a warning. Also note that *htlcmin* only applies to forwarded HTLCs: we can still send smaller payments ourselves.

htlcmax is an optional value that limits how large an HTLC we will forward: if omitted, it is unchanged (the default is no effective limit). 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*. Note that *htlcmax* only applies to forwarded HTLCs: we can still send larger payments ourselves.

enforcedelay is the number of seconds to delay before enforcing the new fees/htlc max (default 600, which is ten minutes). This gives the network a chance to catch up with the new rates and avoids rejecting HTLCs before they do. This only has an effect if rates are increased (we always allow users to overpay fees) or *htlcmax* is decreased, and only applied to a single rate increase per channel (we don't remember an arbitrary number of prior feerates) and if the node is restarted the updated configuration is enforced immediately.

91.3 RETURN VALUE

On success, an object containing **channels** is returned. It is an array of objects, where each object contains:

- **peer_id** (pubkey): The node_id of the peer
- **channel_id** (hex): The channel_id of the channel (always 64 characters)
- **fee_base_msat** (msat): The resulting feebase (this is the BOLT #7 name)
- **fee_proportional_millionths** (u32): The resulting feeppm (this is the BOLT #7 name)
- **minimum_htlc_out_msat** (msat): The resulting htlcmin we will advertize (the BOLT #7 name is htlc_minimum_msat)
- **maximum_htlc_out_msat** (msat): The resulting htlcmax we will advertize (the BOLT #7 name is htlc_maximum_msat)
- **short_channel_id** (short_channel_id, optional): the short_channel_id (if locked in)
- the following warnings are possible:
  - **warning_htlcmin_too_low**: The requested htlcmin was too low for this peer, so we set it to the minimum they will allow
  - **warning_htlcmax_too_high**: The requested htlcmax was greater than the channel capacity, so we set it to the channel capacity

91.4 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.

91.5 AUTHOR

Michael Schmoock <michael@schmoock.net> is the author of this feature. Rusty Russell <rusty@rustcorp.com.au> is mainly responsible for the Core Lightning project.

91.6 SEE ALSO

lightningd-config(5), lightning-fundchannel(7), lightning-listchannels(7), lightning-listpeers(7)

91.7 RESOURCES

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

CHAPTER 92

lightning-setchannelfee – Command for setting specific routing fees on a lightning channel

92.1 SYNOPSIS

(DEPRECATED) **setchannelfee** *id* [*base*] [*ppm*] [*enforcedelay*]

92.2 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, CHANNELD_AWAITING_LOCKIN or DUALOPEND_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, CHANNELD_AWAITING_LOCKIN or DUALOPEND_AWAITING_LOCKIN. If *id* is a peerid, all channels with the peer in those states are changed.

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 through 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.

enforcedelay is the number of seconds to delay before enforcing the new fees (default 600, which is ten minutes). This gives the network a chance to catch up with the new rates and avoids rejecting HTLCs before they do. This only has an effect if rates are increased (we always allow users to overpay fees), only applies to a single rate increase per channel (we don't remember an arbitrary number of prior feerates) and if the node is restarted the updated fees are enforced immediately.

92.3 RETURN VALUE

On success, an object is returned, containing:

- **base** (u32): The fee_base_msat value
- **ppm** (u32): The fee_proportional_millionths value
- **channels** (array of objects): channel(s) whose rate is now set:
  - **peer_id** (pubkey): The node_id of the peer
  - **channel_id** (hex): The channel_id of the channel (always 64 characters)
  - **short_channel_id** (short_channel_id, optional): the short_channel_id (if locked in)

92.4 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.

92.5 AUTHOR

Michael Schmoock <michael@schmoock.net> is the author of this feature. Rusty Russell <rusty@rustcorp.com.au> is mainly responsible for the Core Lightning project.

92.6 SEE ALSO

lightningd-setchannel(7)

92.7 RESOURCES

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

CHAPTER 93

lightning-signmessage – Command to create a signature from this node

93.1 SYNOPSIS

signmessage *message*

93.2 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.

93.3 RETURN VALUE

On success, an object is returned, containing:

- **signature** (hex): The signature (always 128 characters)
- **recid** (hex): The recovery id (0, 1, 2 or 3) (always 2 characters)
- **zbase** (string): *signature* and *recid* encoded in a style compatible with **lnd**'s SignMessageRequest

93.4 AUTHOR

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

93.5 SEE ALSO

lightning-checkmessage(7)

93.6 RESOURCES

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

CHAPTER 94

lightning-signpsbt – Command to sign a wallet's inputs on a provided bitcoin transaction (PSBT).

94.1 SYNOPSIS

signpsbt *psbt* [*signonly*]

94.2 DESCRIPTION

signpsbt is a low-level RPC command which signs a PSBT as defined by BIP-174.

- *psbt*: A string that represents the PSBT value.
- *signonly*: An optional array of input numbers to sign.

By default, all known inputs are signed, and others ignored: with *signonly*, only those inputs are signed, and an error is returned if one of them cannot be signed.

Note that the command will fail if there are no inputs to sign, or if the inputs to be signed were not previously reserved.

94.3 EXAMPLE JSON REQUEST

```
{
  "id": 82,
  "method": "signpsbt",
  "params": {
    "psbt": "some_psbt"
  }
}
```

94.4 RETURN VALUE

On success, an object is returned, containing:

- **signed_psbt** (string): The fully signed PSBT

On failure, one of the following error codes may be returned:

- -32602: Error in given parameters, or there aren't wallet's inputs to sign, or we couldn't sign all of *signonly*, or inputs are not reserved.

94.5 EXAMPLE JSON RESPONSE

```
{
    "psbt": "some_psbt"
}
```

94.6 AUTHOR

Vincenzo Palazzo <vincenzo.palazzo@protonmail.com> wrote the initial version of this man page, but many others did the hard work of actually implementing this rpc command.

94.7 SEE ALSO

lightning-fundpsbt(7), lightning-sendpsbt(7)

94.8 RESOURCES

Main web site: https://github.com/ElementsProject/lightning *comment*: # (SHA256STAMP:0daef100b12490126849fcb93a9e861554807d1a5acf68bc17de92a30505e18a)

CHAPTER 95

lightning-staticbackup – Command for deriving getting SCB of all the existing channels

95.1 SYNOPSIS

staticbackup

95.2 DESCRIPTION

The **staticbackup** RPC command returns an object with SCB of all the channels in an array.

95.3 RETURN VALUE

On success, an object is returned, containing:

- **scb** (array of hexs):
  - Each item is SCB of a channel in TLV format

95.4 AUTHOR

Aditya <aditya.sharma20111@gmail.com> is mainly responsible.

95.5 SEE ALSO

lightning-getsharedsecret(7)

95.6 RESOURCES

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

CHAPTER 96

lightning-stop – Command to shutdown the Core Lightning node.

96.1 SYNOPSIS

stop

96.2 DESCRIPTION

The **stop** is a RPC command to shut off the Core Lightning node.

96.3 EXAMPLE JSON REQUEST

```
{
  "id": 82,
  "method": "stop",
  "params": {}
}
```

96.4 RETURN VALUE

On success, returns a single element (string) (always "Shutdown complete") *comment*: # (GENERATE-FROM-SCHEMA-END)

Once it has returned, the daemon has cleaned up completely, and if desired may be restarted immediately.

96.5 AUTHOR

Vincenzo Palazzo <vincenzo.palazzo@protonmail.com> wrote the initial version of this man page, but many others did the hard work of actually implementing this rpc command.

96.6 RESOURCES

Main web site: https://github.com/ElementsProject/lightning *comment*: # (SHA256STAMP:2103952683449a5aa313eefa9c850dc0ae1cf4aa65edeb7897a8748a010a9349)

CHAPTER 97

lightning-txdiscard – Abandon a transaction from txprepare, release inputs

97.1 SYNOPSIS

txdiscard *txid*

97.2 DESCRIPTION

The **txdiscard** RPC command releases inputs which were reserved for use of the *txid* from lightning-txprepare(7).

97.3 RETURN VALUE

On success, an object is returned, containing:

- **unsigned_tx** (hex): the unsigned transaction
- **txid** (txid): the transaction id of *unsigned_tx*

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*.

The following error codes may occur:

- -1: An unknown *txid*.

97.4 AUTHOR

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

97.5 SEE ALSO

lightning-txprepare(7), lightning-txsend(7)

97.6 RESOURCES

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

CHAPTER 98

lightning-txprepare – Command to prepare to withdraw funds from the internal wallet

98.1 SYNOPSIS

txprepare *outputs* [*feerate*] [*minconf*] [*utxos*]

98.2 DESCRIPTION

The **txprepare** RPC command creates an unsigned transaction which spends funds from Core 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 any number of **confirmed** 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**.

98.3 RETURN VALUE

On success, an object is returned, containing:

- **psbt** (string): the PSBT representing the unsigned transaction
- **unsigned_tx** (hex): the unsigned transaction
- **txid** (txid): the transaction id of *unsigned_tx*; you hand this to lightning-txsend(7) or lightning-txdiscard(7), as the inputs of this transaction are reserved.

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.

98.4 AUTHOR

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

98.5 SEE ALSO

lightning-withdraw(7), lightning-txsend(7), lightning-txdiscard(7), lightning-feerates(7)

98.6 RESOURCES

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

CHAPTER 99

lightning-txsend – Command to sign and send transaction from txprepare

99.1 SYNOPSIS

txsend *txid*

99.2 DESCRIPTION

The **txsend** RPC command signs and broadcasts a transaction created by **txprepare**.

99.3 RETURN VALUE

On success, an object is returned, containing:

- **psbt** (string): the completed PSBT representing the signed transaction
- **tx** (hex): the fully signed transaction
- **txid** (txid): the transaction id of *tx*

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.

99.4 AUTHOR

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

99.5 SEE ALSO

lightning-txprepare(7), lightning-txdiscard(7)

99.6 RESOURCES

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

CHAPTER 100

lightning-unreserveinputs – Release reserved UTXOs

100.1 SYNOPSIS

unreserveinputs *psbt* [*reserve*]

100.2 DESCRIPTION

The **unreserveinputs** RPC command releases (or reduces reservation) on UTXOs which were previously marked as reserved, generally by lightning-reserveinputs(7).

The inputs to unreserve are the inputs specified in the passed-in *psbt*.

If *reserve* is specified, it is the number of blocks to decrease reservation by; default is 72.

100.3 RETURN VALUE

On success, an object containing **reservations** is returned. It is an array of objects, where each object contains:

- **txid** (txid): the transaction id
- **vout** (u32): the output number which was reserved
- **was_reserved** (boolean): whether the input was already reserved (usually `true`)
- **reserved** (boolean): whether the input is now reserved (may still be `true` if it was reserved for a long time)

If **reserved** is *true*:

- **reserved_to_block** (u32): what blockheight the reservation will expire

On failure, an error is reported and no UTXOs are unreserved.

The following error codes may occur:

- -32602: Invalid parameter, i.e. an unparseable PSBT.

100.4 AUTHOR

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

100.5 SEE ALSO

lightning-unreserveinputs(7), lightning-signpsbt(7), lightning-sendpsbt(7)

100.6 RESOURCES

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

CHAPTER 101

lightning-utxopsbt – Command to populate PSBT inputs from given UTXOs

101.1 SYNOPSIS

utxopsbt *satoshi* *feerate* *startweight* *utxos* [*reserve*] [*reservedok*] [*locktime*] [*min_witness_weight*] [*excess_as_change*]

101.2 DESCRIPTION

utxopsbt is a low-level RPC command which creates a PSBT using unreserved inputs in the wallet, optionally reserving them as well.

It deliberately mirrors the parameters and output of lightning-fundpsbt(7) except instead of an optional *minconf* parameter to select unreserved outputs from the wallet, it takes a compulsory list of outputs to use.

utxos must be an array of "txid:vout", each of which must be reserved or available: the total amount must be sufficient to pay for the resulting transaction plus *startweight* at the given *feerate*, with at least *satoshi* left over (unless *satoshi* is **all**, which is equivalent to setting it to zero).

If *reserve* if not zero, then *reserveinputs* is called (successfully, with *exclusive* true) on the returned PSBT for this number of blocks (default 72 blocks if unspecified).

Unless *reservedok* is set to true (default is false) it will also fail if any of the *utxos* are already reserved.

locktime is an optional locktime: if not set, it is set to a recent block height.

min_witness_weight is an optional minimum weight to use for a UTXO's witness. If the actual witness weight is greater than the provided minimum, the actual witness weight will be used.

excess_as_change is an optional boolean to flag to add a change output for the excess sats.

101.3 RETURN VALUE

On success, an object is returned, containing:

- **psbt** (string): Unsigned PSBT which fulfills the parameters given
- **feerate_per_kw** (u32): The feerate used to create the PSBT, in satoshis-per-kiloweight
- **estimated_final_weight** (u32): The estimated weight of the transaction once fully signed
- **excess_msat** (msat): The amount above *satoshi* which is available. This could be zero, or dust; it will be zero if *change_outnum* is also returned
- **change_outnum** (u32, optional): The 0-based output number where change was placed (only if parameter *excess_as_change* was true and there was sufficient funds)
- **reservations** (array of objects, optional): If *reserve* was true or a non-zero number, just as per lightning-reserveinputs(7):
  - **txid** (txid): The txid of the transaction
  - **vout** (u32): The 0-based output number
  - **was_reserved** (boolean): Whether this output was previously reserved
  - **reserved** (boolean): Whether this output is now reserved (always *true*)
  - **reserved_to_block** (u32): The blockheight the reservation will expire

On success, returns the *psbt* it created, containing the inputs, *feerate_per_kw* showing the exact numeric feerate it used, *estimated_final_weight* for the estimated weight of the transaction once fully signed, and *excess_msat* containing the amount above *satoshi* which is available. This could be zero, or dust. If *satoshi* was "all", then *excess_msat* is the entire amount once fees are subtracted for the weights of the inputs and *startweight*.

If *reserve* was *true* or a non-zero number, then a *reservations* array is returned, exactly like *reserveinputs*.

If *excess_as_change* is true and the excess is enough to cover an additional output above the `dust_limit`, then an output is added to the PSBT for the excess amount. The *excess_msat* will be zero. A *change_outnum* will be returned with the index of the change output.

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.
- 301: Insufficient UTXOs to meet *satoshi* value.

101.4 AUTHOR

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

101.5 SEE ALSO

lightning-fundpsbt(7).

101.6 RESOURCES

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

CHAPTER 102

lightning-waitanyinvoice – Command for waiting for payments

102.1 SYNOPSIS

waitanyinvoice [*lastpay_index*] [*timeout*]

102.2 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.

If *timeout* is specified, wait at most that number of seconds, which must be an integer. If the specified *timeout* is reached, this command will return with an error. You can specify this to 0 so that **waitanyinvoice** will return immediately with an error if no pending invoice is available yet. If unspecified, this command will wait indefinitely.

102.3 RETURN VALUE

On success, an object is returned, containing:

- **label** (string): unique label supplied at invoice creation
- **description** (string): description used in the invoice
- **payment_hash** (hash): the hash of the *payment_preimage* which will prove payment (always 64 characters)
- **status** (string): Whether it's paid or expired (one of "paid", "expired")
- **expires_at** (u64): UNIX timestamp of when it will become / became unpayable

- **amount_msat** (msat, optional): the amount required to pay this invoice
- **bolt11** (string, optional): the BOLT11 string (always present unless *bolt12* is)
- **bolt12** (string, optional): the BOLT12 string (always present unless *bolt11* is)

If **status** is "paid":

- **pay_index** (u64): Unique incrementing index for this payment
- **amount_received_msat** (msat): the amount actually received (could be slightly greater than *amount_msat*, since clients may overpay)
- **paid_at** (u64): UNIX timestamp of when it was paid
- **payment_preimage** (secret): proof of payment (always 64 characters)

Possible errors are:

- 1.

  The *timeout* was reached without an invoice being paid.

102.4 AUTHOR

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

102.5 SEE ALSO

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

102.6 RESOURCES

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

CHAPTER 103

lightning-waitblockheight – Command for waiting for blocks on the blockchain

103.1 SYNOPSIS

waitblockheight *blockheight* [*timeout*]

103.2 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.

103.3 RETURN VALUE

On success, an object is returned, containing:

- **blockheight** (u32): The current block height (>= *blockheight* parameter)

If *timeout* seconds is reached without the specified blockheight being reached, this command will fail with a code of `2000`.

103.4 AUTHOR

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

103.5 RESOURCES

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

CHAPTER 104

lightning-waitinvoice – Command for waiting for specific payment

104.1 SYNOPSIS

waitinvoice *label*

104.2 DESCRIPTION

The **waitinvoice** RPC command waits until a specific invoice is paid, then returns that single entry as per **listinvoice**.

104.3 RETURN VALUE

On success, an object is returned, containing:

- **label** (string): unique label supplied at invoice creation
- **description** (string): description used in the invoice
- **payment_hash** (hash): the hash of the *payment_preimage* which will prove payment (always 64 characters)
- **status** (string): Whether it's paid or expired (one of "paid", "expired")
- **expires_at** (u64): UNIX timestamp of when it will become / became unpayable
- **amount_msat** (msat, optional): the amount required to pay this invoice
- **bolt11** (string, optional): the BOLT11 string (always present unless *bolt12* is)
- **bolt12** (string, optional): the BOLT12 string (always present unless *bolt11* is)

If **status** is "paid":

- **pay_index** (u64): Unique incrementing index for this payment

- **amount_received_msat** (msat): the amount actually received (could be slightly greater than *amount_msat*, since clients may overpay)
- **paid_at** (u64): UNIX timestamp of when it was paid
- **payment_preimage** (secret): proof of payment (always 64 characters)

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: If the invoice is deleted while unpaid, or the invoice does not exist.
- 903: If the invoice expires before being paid, or is already expired.

104.4 AUTHOR

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

104.5 SEE ALSO

lightning-waitanyinvoice(7), lightning-listinvoice(7), lightning-delinvoice(7), lightning-invoice(7)

104.6 RESOURCES

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

CHAPTER 105

lightning-waitsendpay – Command for sending a payment via a route

105.1 SYNOPSIS

waitsendpay *payment_hash* [*timeout*] [*partid*]

105.2 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.

105.3 RETURN VALUE

On success, an object is returned, containing:

- **id** (u64): unique ID for this payment attempt
- **payment_hash** (hash): the hash of the *payment_preimage* which will prove payment (always 64 characters)
- **status** (string): status of the payment (always "complete")
- **created_at** (u64): the UNIX timestamp showing when this payment was initiated

- **amount_sent_msat** (msat): The amount sent
- **groupid** (u64, optional): Grouping key to disambiguate multiple attempts to pay an invoice or the same payment_hash
- **amount_msat** (msat, optional): The amount delivered to destination (if known)
- **destination** (pubkey, optional): the final destination of the payment if known
- **completed_at** (number, optional): the UNIX timestamp showing when this payment was completed
- **label** (string, optional): the label, if given to sendpay
- **partid** (u64, optional): the *partid*, if given to sendpay
- **bolt11** (string, optional): the bolt11 string (if pay supplied one)
- **bolt12** (string, optional): the bolt12 string (if supplied for pay: **experimental-offers** only).

If **status** is "complete":

- **payment_preimage** (secret): the proof of payment: SHA256 of this **payment_hash** (always 64 characters)

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.

105.4 AUTHOR

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

105.5 SEE ALSO

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

105.6 RESOURCES

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

CHAPTER 106

lightning-withdraw – Command for withdrawing funds from the internal wallet

106.1 SYNOPSIS

withdraw *destination satoshi* [*feerate*] [*minconf*] [*utxos*]

106.2 DESCRIPTION

The **withdraw** RPC command sends funds from Core 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.

106.3 RETURN VALUE

On success, an object is returned, containing:

- **tx** (hex): the fully signed bitcoin transaction
- **txid** (txid): the transaction id of *tx*
- **psbt** (string): the PSBT representing the unsigned transaction

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.

106.4 AUTHOR

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

106.5 SEE ALSO

lightning-listfunds(7), lightning-fundchannel(7), lightning-newaddr(7), lightning-txprepare(7), lightning-feerates(7).

106.6 RESOURCES

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

CHAPTER 107

lightningd – Daemon for running a Lightning Network node

107.1 SYNOPSIS

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

107.2 DESCRIPTION

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

107.3 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 Core Lightning daemon. Defaults to *$HOME/.lightning*.

107.4 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.

107.5 LOGGING AND COMMANDING CORE LIGHTNING

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

Core 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 Core Lightning once **lightningd** has started; you need to match the **–lightning-dir** and **–rpc-file** options between them.

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

107.6 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: otherwise indicate *network=testnet* in your *config* file explicitly.

Core 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 at that IP:

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

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
```

107.7 ERRORS CODE

- 1: Generic lightning-cli error
- 10: Error executing subdaemons
- 11: Error locking pidfile (often another lightningd running)
- 20: Generic error related to HSM secret
- 21: HSM secret is encrypted
- 22: Bad password used to decrypt the HSM secred
- 23: Error caused from the I/O operation during a HSM decryption/encryption operation
- 30: Wallet database does not match (network or hsm secret)

107.8 BUGS

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

107.9 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.

107.10 SEE ALSO

lightningd-rpc(7), lightning-listconfigs(7), lightningd-config(5), lightning-cli(1), lightning-newaddr(7), lightning-listfunds(7), lightning-connect(7), lightning-fundchannel(7), lightning-listpeers(7), lightning-pay(7), lightning-hsmtool(8)

107.11 RESOURCES

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

107.12 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 108

lightningd-config – Lightning daemon configuration file

108.1 SYNOPSIS

~/.lightning/config

108.2 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.

108.3 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.

108.4 OPTIONS

108.4.1 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.

- **database-upgrade**=*BOOL*

  Upgrades to Core Lightning often change the database: once this is done, downgrades are not generally possible. By default, Core Lightning will exit with an error rather than upgrade, unless this is an official released version. If you really want to upgrade to a non-release version, you can set this to *true* (or *false* to never allow a non-reversible upgrade!).

108.4.2 Bitcoin control options:

Bitcoin control options:

- **network**=*NETWORK*

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

- **mainnet**

  Alias for *network=bitcoin*.

- **testnet**

  Alias for *network=testnet*.

- **signet**

  Alias for *network=signet*.

- **bitcoin-cli**=*PATH* [plugin `bcli`]

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

- **bitcoin-datadir**=*DIR* [plugin `bcli`]

  -datadir argument to supply to bitcoin-cli(1).

- **bitcoin-rpcuser**=*USER* [plugin `bcli`]

  The RPC username for talking to bitcoind(1).

- **bitcoin-rpcpassword**=*PASSWORD* [plugin `bcli`]

  The RPC password for talking to bitcoind(1).

- **bitcoin-rpcconnect**=*HOST* [plugin `bcli`]

  The bitcoind(1) RPC host to connect to.

- **bitcoin-rpcport**=*PORT* [plugin `bcli`]

  The bitcoind(1) RPC port to connect to.

- **bitcoin-retry-timeout**=*SECONDS* [plugin `bcli`]

  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.

108.4.3 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*.

- **subdaemon**=*SUBDAEMON*:*PATH*

  Specifies an alternate subdaemon binary. Current subdaemons are *channeld*, *closingd*, *connectd*, *gossipd*, *hsmd*, *onchaind*, and *openingd*. If the supplied path is relative the subdaemon binary is found in the working directory. This option may be specified multiple times.

  So, **subdaemon=hsmd:remote_signer** would use a hypothetical remote signing proxy instead of the standard *lightning_hsmd* binary.

- **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 (or *nodeid*) containing that string. This option may be specified multiple times. 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-level=debug:024b9a1fa8** would set debug logging for that channel (or any node id containing that string).

- **log-prefix**=*PREFIX*

  Prefix for all log lines: this can be customized if you want to merge logs with multiple daemons. Usually you want to include a space at the end of *PREFIX*, as the timestamp follows immediately.

- **log-file**=*PATH*

  Log to this file (instead of stdout). If you specify this more than once you'll get more than one log file: **-** is used to mean stdout. Sending lightningd(8) SIGHUP will cause it to reopen each file (useful for log rotation).

- **log-timestamps**=*BOOL*

  Set this to false to turn off timestamp prefixes (they will still appear in crash log files).

- **rpc-file**=*PATH*

  Set JSON-RPC socket (or /dev/tty), such as for lightning-cli(1).

- **rpc-file-mode**=*MODE*

  Set JSON-RPC socket file mode, as a 4-digit octal number. Default is 0600, meaning only the user that launched lightningd can command it. Set to 0660 to allow users with the same group to access the RPC as well.

- **daemon**

  Run in the background, suppress stdout and stderr. Note that you need to specify **log-file** for this case.

- **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.

  The default wallet corresponds to the following DSN: `--wallet=sqlite3://$HOME/.lightning/bitcoin/lightningd.sqlite31`

  For the `sqlite3` scheme, you can specify a single backup database file by separating it with a `:` character, like so: `--wallet=sqlite3://$HOME/.lightning/bitcoin/lightningd.sqlite3:/backup/lightningd.sqlite3`

  The following is an example of a postgresql wallet DSN:

  `--wallet=postgres://user:pass@localhost:5432/db_name`

  This will connect to a DB server running on `localhost` port `5432`, authenticate with username `user` and password `pass`, and then use the database `db_name`. The database must exist, but the schema will be managed automatically by `lightningd`.

- **bookkeeper-dir**=*DIR* [plugin `bookkeeper`]

  Directory to keep the accounts.sqlite3 database file in. Defaults to lightning-dir.

- **bookkeeper-db**=*DSN* [plugin `bookkeeper`]

  Identify the location of the bookkeeper data. This is a fully qualified data source name, including a scheme such as `sqlite3` or `postgres` followed by the connection parameters. Defaults to `sqlite3://accounts.sqlite3` in the `bookkeeper-dir`.

- **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. If there is no `hsm_secret` yet, `lightningd` will create a new encrypted secret. If you have an unencrypted `hsm_secret` you want to encrypt on-disk, or vice versa, see lightning-hsmtool(8).

- **grpc-port**=*portnum* [plugin `cln-grpc`]

  The port number for the GRPC plugin to listen for incoming connections; default is not to activate the plugin at all.

108.4.4 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-setchannel(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-setchannel(7).

- **min-capacity-sat**=*SATOSHI*

  Default: 10000. This value defines the minimal effective channel capacity in satoshi to accept for channel opening requests. This will reject any opening of a channel which can't pass an HTLC of least this value. Usually this prevents a peer opening a tiny channel, but it can also prevent a channel you open with a reasonable amount and the peer requesting such a large reserve that the capacity of the channel falls below this.

- **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.

- **force-feerates**==*VALUES*

  Networks like regtest and testnet have unreliable fee estimates: we usually treat them as the minimum (253 sats/kw) if we can't get them. This allows override of one or more of our standard feerates (see lightning-feerates(7)). Up to 5 values, separated by '/' can be provided: if fewer are provided, then the final value is used for the remainder. The values are in per-kw (roughly 1/4 of bitcoind's per-kb values), and the order is "opening", "mutual_close", "unilateral_close", "delayed_to_us", "htlc_resolution", and "penalty".

  You would usually put this option in the per-chain config file, to avoid setting it on Bitcoin mainnet! e.g. `~rusty/.lightning/regtest/config`.

- **htlc-minimum-msat**=*MILLISATOSHI*

  Default: 0. Sets the minimal allowed HTLC value for newly created channels. If you want to change the `htlc_minimum_msat` for existing channels, use the RPC call lightning-setchannel(7).

- **htlc-maximum-msat**=*MILLISATOSHI*

  Default: unset (no limit). Sets the maximum allowed HTLC value for newly created channels. If you want to change the `htlc_maximum_msat` for existing channels, use the RPC call lightning-setchannel(7).

- **disable-ip-discovery**

  Turn off public IP discovery to send `node_announcement` updates that contain the discovered IP with TCP port 9735 as announced address. If unset and you open TCP port 9735 on your router towords your node, your node will remain connectable on changing IP addresses. Note: Will always be disabled if you use 'always-use-proxy'.

108.4.5 Lightning channel and HTLC options

- **large-channels**

  Removes capacity limits for channel creation. Version 1.0 of the specification limited channel sizes to 16777215 satoshi. With this option (which your node will advertize to peers), your node will accept larger incoming channels and if the peer supports it, will open larger channels. Note: this option is spelled **large-channels** but it's pronounced **wumbo**.

- **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* [plugin `bcli`]

  The percentage of *estimatesmartfee 2/CONSERVATIVE* to use for the commitment transactions: default is 100.

- **max-concurrent-htlcs**=*INTEGER*

  Number of HTLCs one channel can handle concurrently in each direction. Should be between 1 and 483 (default 30).

- **max-dust-htlc-exposure-msat**=*MILLISATOSHI*

  Option which limits the total amount of sats to be allowed as dust on a channel.

- **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.

- **accept-htlc-tlv-types**=*types*

  Normally HTLC onions which contain unknown even fields are rejected. This option specifies that these (comma-separated) types are to be accepted, and ignored.

108.4.6 Cleanup control options:

- **autoclean-cycle**=*SECONDS* [plugin `autoclean`]

  Perform search for things to clean every *SECONDS* seconds (default 3600, or 1 hour, which is usually sufficient).

- **autoclean-succeededforwards-age**=*SECONDS* [plugin `autoclean`]

  How old successful forwards (`settled` in listforwards `status`) have to be before deletion (default 0, meaning never).

- **autoclean-failedforwards-age**=*SECONDS* [plugin `autoclean`]

  How old failed forwards (`failed` or `local_failed` in listforwards `status`) have to be before deletion (default 0, meaning never).

- **autoclean-succeededpays-age**=*SECONDS* [plugin `autoclean`]

  How old successful payments (`complete` in listpays `status`) have to be before deletion (default 0, meaning never).

- **autoclean-failedpays-age**=*SECONDS* [plugin `autoclean`]

  How old failed payment attempts (`failed` in listpays `status`) have to be before deletion (default 0, meaning never).

- **autoclean-paidinvoices-age**=*SECONDS* [plugin `autoclean`]

  How old invoices which were paid (`paid` in listinvoices `status`) have to be before deletion (default 0, meaning never).

- **autoclean-expiredinvoices-age**=*SECONDS* [plugin `autoclean`]

  How old invoices which were not paid (and cannot be) (`expired` in listinvoices `status`) before deletion (default 0, meaning never).

Note: prior to v22.11, forwards for channels which were closed were not easily distinguishable. As a result, autoclean may delete more than one of these at once, and then suffer failures when it fails to delete the others.

108.4.7 Payment control options:

- **disable-mpp** [plugin `pay`]

  Disable the multi-part payment sending support in the `pay` plugin. By default the MPP support is enabled, but it can be desirable to disable in situations in which each payment should result in a single HTLC being forwarded in the network.

108.4.8 Networking options

Note that for simple setups, the implicit *autolisten* option does the right thing: for the mainnet (bitcoin) network 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 (and other default ports for other networks, as described below).

Core Lightning also support IPv4/6 address discovery behind NAT routers. If your node detects an new public address, it will update its announcement. For this to work you need to forward the default TCP port 9735 to your node. IP discovery is only active if no other addresses are announced.

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[:SE*

  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 you want to specify an IPv6 address *and* a port, use `[]` around the IPv6 address, like `[::]:9750`). If 'PORT' is not specified, the default port 9735 is used for mainnet (testnet: 19735, signet: 39735, regtest: 19846). If we can determine a public IP address from the resulting binding, the address is announced.

  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 in case of mainnet (bitcoin) network whereas other networks (testnet, signet, regtest) will set the same default ports they use for non-Tor addresses (see above). 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 node's id.

  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. 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 possibly 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.

  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). The port defaults to 9050 if not specified.

- **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.

- **tor-service-password**=*PASSWORD*

  Set a Tor control password, which may be needed for *autotor:* to authenticate to the Tor control port.

108.4.9 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 Core Lightning. This can be specified multiple times to add multiple plugins. Note that unless plugins themselves specify ordering requirements for being called on various hooks, plugins will be ordered by commandline, then config file.

- **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. The ordering of plugins within a directory is currently unspecified.

- **clear-plugins**

  This option clears all *plugin*, *important-plugin*, and *plugin-dir* options preceeding it, including the default built-in plugin directory. You can still add *plugin-dir*, *plugin*, and *important-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* will not be loaded at startup. Otherwise, no plugin with that base name will be loaded at startup, whatever directory it is in. This option is useful for disabling a single plugin inside a directory. You can still explicitly load plugins which have been disabled, using lightning-plugin(7) `start`.

- **important-plugin**=*PLUGIN*

  Speciy a plugin to run as part of Core Lightning. This can be specified multiple times to add multiple plugins. Plugins specified via this option are considered so important, that if the plugin stops for any reason (including via lightning-plugin(7) `stop`), Core Lightning will also stop running. This way, you can monitor crashes of important plugins by simply monitoring if Core Lightning terminates. Built-in plugins, which are installed with lightningd(8), are automatically considered important.

108.4.10 Experimental Options

Experimental options are subject to breakage between releases: they are made available for advanced users who want to test proposed features. When the build is configured *without* `--enable-experimental-features`, below options are available but disabled by default. A build *with* `--enable-experimental-features` enables some of below options by default and also adds support for even more features. Supported features can be listed with `lightningd --list-features-only`.

- **experimental-onion-messages**

  Specifying this enables sending, forwarding and receiving onion messages, which are in draft status in the BOLT specifications.

- **experimental-offers**

  Specifying this enables the `offers` and `fetchinvoice` plugins and corresponding functionality, which are in draft status as BOLT12. This usually requires **experimental-onion-messages** as well. See lightning-offer(7) and lightning-fetchinvoice(7).

- **fetchinvoice-noconnect**

  Specifying this prevents `fetchinvoice` and `sendinvoice` from trying to connect directly to the offering node as a last resort.

- **experimental-shutdown-wrong-funding**

  Specifying this allows the `wrong_funding` field in shutdown: if a remote node has opened a channel but claims it used the incorrect txid (and the channel hasn't been used yet at all) this allows them to negotiate a clean shutdown with the txid they offer.

- **experimental-dual-fund**

  Specifying this enables support for the dual funding protocol, allowing both parties to contribute funds to a channel. The decision about whether to add funds or not to a proposed channel is handled automatically by a plugin that implements the appropriate logic for your needs. The default behavior is to not contribute funds.

- **experimental-websocket-port**=*PORT*

  Specifying this enables support for accepting incoming WebSocket connections on that port, on any IPv4 and IPv6 addresses you listen to. The normal protocol is expected to be sent over WebSocket binary frames once the connection is upgraded.

108.5 BUGS

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

108.6 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.

108.7 SEE ALSO

lightning-listconfigs(7) lightning-setchannel(7) lightningd(8) lightning-hsmtool(8)

108.8 RESOURCES

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

108.9 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 109

lightningd-rpc – Lightning Daemon RPC Protocols

109.1 SYNOPSIS

~/.lightning/bitcoin/lightning-rpc

109.2 DESCRIPTION

lightningd(8) communicates via RPC, especially JSONRPC over the UNIX domain socket (by default **$HOME/.lightning/bitcoin/lightning-rpc**, but configuable with lightningd-config(5)).

109.3 JSON WIRE FORMAT

JSON RPC is defined at https://www.jsonrpc.org/specification and generally involves writing a JSON request with a unique ID, and receiving a response containing that ID.

Every response given by lightningd(8) is followed by two '\n' characters, which should not appear in normal JSON (though plugins may produce them). This means efficient code can simply read until it sees two '\n' characters, and then attempt to parse the JSON (if the JSON is incomplete, it should continue reading and file a bug).

109.4 JSON COMMANDS

We support "params" as an array (ordered parameters) or a dictionary (named parameters). In the array case, JSON "null" is treated as if the parameter was not specified (if that is allowed).

You should probably prefer named parameters if possible, as they have generally been shown to be less confusing for complex commands and more robust when fields are deprecated.

The lightning-cli(1) tool uses ordered parameters by default, but named parameters if explicitly specified or the first parameter contains an '='.

109.5 JSON REPLIES

All JSON replies are wrapped in an object; this allows fields to be added in future. You should safely ignore any unknown fields.

Any field name which starts with "warning" is a specific warning, and should be documented in the commands' manual page. Each warning field has an associated human-readable string, but it's redudant, as each separate warning should have a distinct field name (e.g. **warning_offer_unknown_currency** and **warning_offer_missing_description**).

109.6 JSON TYPES

The exact specification for (most!) commands is specified in `doc/schemas/` in the source directory. This is also used to generate part of the documentation for each command; the following types are referred to in addition to simple JSON types:

- `hex`: an even-length string of hexidecimal digits.
- `hash`: a 64-character `hex` which is a sha256 hash.
- `secret`: a 64-character `hex` which is a secret of some kind.
- `u64`: a JSON number without decimal point in the range 0 to 18446744073709551615 inclusive.
- `u32`: a JSON number without decimal point in the range 0 to 4294967295 inclusive.
- `u16`: a JSON number without decimal point in the range 0 to 65535 inclusive.
- `u16`: a JSON number without decimal point in the range 0 to 255 inclusive.
- `pubkey`: a 66-character `hex` which is an SEC-1 encoded secp256k1 point (usually used as a public key).
- `msat`: a `u64` which indicates an amount of millisatoshis. Deprecated: may also be a string of the number, with "msat" appended. As an input parameter, lightningd(8) will accept strings with suffixes (see below).
- `txid`: a 64-character `hex` Bitcoin transaction identifier.
- `signature`: a `hex` (144 bytes or less), which is a DER-encoded Bitcoin signature (without any sighash flags appended),
- `bip340sig`: a 128-character `hex` which is a BIP-340 (Schnorr) signature.
- `point32`: a 64-character `hex` which represents an x-only pubkey.
- `short_channel_id`: a string of form BLOCK "x" TXNUM "x" OUTNUM.
- `short_channel_id_dir`: a `short_channel_id` with "/0" or "/1" appended, indicating the direction between peers.
- `outpoint`: a string containing a `txid` followed by a ":" and an output number (bitcoind uses this form).
- `feerate`: an integer, or a string consisting of a number followed by "perkw" or "perkb".
- `outputdesc`: an object containing onchain addresses as keys, and "all" or a valid `msat` field as values.

The following forms of `msat` are supported as parameters:

- An integer (representing that many millisatoshis), e.g. `10000`
- A string of an integer N and the suffix *msat* (representing N millisatoshis) e.g. `"10000msat"`
- A string of an integer N and the suffix *sat* (representing N times 1000 millisatoshis) e.g. `"10sat"`
- A string of a number N.M (where M is exactly three digits) and the suffix *sat* (representing N times 1000 plus M millisatoshis) e.g. `"10.000sat"`

- A string of an integer N and the suffix *btc* (representing N times 100000000000 millisatoshis) e.g. `"1btc"`
- A string of a number N.M (where M is exactly eight digits) and the suffix *btc* (representing N times 100000000000 plus M times 1000 millisatoshis) e.g `"0.00000010btc"`
- A string of a number N.M (where M is exactly elevent digits) and the suffix *btc* (representing N times 100000000000 plus M millisatoshis) e.g `"0.00000010000btc"`

109.7 JSON NOTIFICATIONS

Notifications are (per JSONRPC spec) JSON commands without an "id" field. They give information about ongoing commands, but you need to enable them. See lightning-notifications(7).

109.8 FIELD FILTERING

You can restrict what fields are in the output of any command, by including a `"filter"` member in your request, alongside the standard `"method"` and `"params"` fields.

`filter` is a template, with keys indicating what fields are to be output (values must be `true`). Only fields which appear in the template will be output. For example, here is a normal `result` of `listtransactions`:

```
"result": {
  "transactions": [
    {
      "hash": "3b15dbc81d6a70abe1e75c1796c3eeba71c3954b7a90dfa67d55c1e989e20dbb",
      "rawtx":
↪"020000000001019db609b099735fada240b82cec9da880b35d7a944065c280b8534cb4e2f5a7e90000000000feffffff0
↪",
      "blockheight": 102,
      "txindex": 1,
      "locktime": 101,
      "version": 2,
      "inputs": [
        {
          "txid": "e9a7f5e2b44c53b880c26540947a5db380a89dec2cb840a2ad5f7399b009b69d",
          "index": 0,
          "sequence": 4294967294
        }
      ],
      "outputs": [
        {
          "index": 0,
          "amount_msat": "1000000000msat",
          "type": "deposit",
          "scriptPubKey": "a914d8b7ebd0ccc80266a97d9a828baf1877032ac66487"
        },
        {
          "index": 1,
          "amount_msat": "4998999857000msat",
          "scriptPubKey": "a9142cb0814338091a73b388579b025c34f328dfb78987"
        }
      ]
    },
    {
```

(continues on next page)

(continued from previous page)

```
      "hash": "3a5ebaae466a9cb69c59553a3100ed545523e7450c32684cbc6bf0b305a6c448",
      "rawtx":
↪"02000000000101bb0de289e9c1557da6df907a4b95c371baeec396175ce7e1ab706a1dc8db153b000000001716001401fa
↪",
      "blockheight": 103,
      "txindex": 1,
      "locktime": 102,
      "version": 2,
      "inputs": [
        {
          "txid": "3b15dbc81d6a70abe1e75c1796c3eeba71c3954b7a90dfa67d55c1e989e20dbb",
          "index": 0,
          "sequence": 4294967293
        }
      ],
      "outputs": [
        {
          "index": 0,
          "amount_msat": "894743000msat",
          "type": "deposit",
          "scriptPubKey": "0014c2ccab171c2a5be9dab52ec41b825863024c5466"
        },
        {
          "index": 1,
          "amount_msat": "100000000msat",
          "type": "channel_funding",
          "channel": "103x1x1",
          "scriptPubKey":
↪"00205b8cd3b914cf67cdd8fa6273c930353dd36476734fbd962102c2df53b90880cd"
        }
      ]
    }
  ]
}
```

If we only wanted the output amounts and types, we would create a filter like so:

```
"filter": {"transactions": [{"outputs": [{"amount_msat": true, "type": true}]}]}
```

The result would be:

```
"result": {
  "transactions": [
    {
      "outputs": [
        {
          "amount_msat": "1000000000msat",
          "type": "deposit",
        },
        {
          "amount_msat": "4998999857000msat",
        }
      ]
    },
    {
      "outputs": [
        {
```

(continues on next page)

(continued from previous page)

```
          "amount_msat": "894743000msat",
          "type": "deposit",
        },
        {
          "amount_msat": "100000000msat",
          "type": "channel_funding",
        }
      ]
    }
  ]
}
```

Note: `"filter"` doesn't change the order, just which fields are printed. Any fields not explictly mentioned are omitted from the output, but plugins which don't support filter (and some routines doing simple JSON transfers) may ignore `"filter"`, so you should treat it as an optimazation only).

Note: if you specify an array where an object is specified or vice versa, the response may include a `warning_parameter_filter` field which describes the problem.

109.9 DEALING WITH FORMAT CHANGES

Fields can be added to the JSON output at any time, but to remove (or, very rarely) change a field requires a minimum deprecation period of 6 months and two releases. Usually a new field will be added if one is deprecated, so both will be present in transition.

To test that you're not using deprecated fields, you can use the lightningd-config(5) option `allow-deprecated-apis=false`. You should only use this in internal tests: it is not recommended that users use this directly.

The documentation tends to only refer to non-deprecated items, so if you seen an output field which is not documented, its either a bug (like that ever happens!) or a deprecated field you should ignore.

109.10 DEBUGGING

You can use `log-level=io` to see much of the JSON conversation (in hex) that occurs. It's extremely noisy though!

109.11 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.

109.12 SEE ALSO

lightningd-config(5), lightning-notifications(7), lightningd(8)

109.13 RESOURCES

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

109.14 COPYING