btcd

Release beta

Sep 01, 2020

Contents

build passing Build Status ISC License GoDoc

btcd is an alternative full node bitcoin implementation written in Go (golang).

This project is currently under active development and is in a Beta state. It is extremely stable and has been in production use since October 2013.

It properly downloads, validates, and serves the block chain using the exact rules (including consensus bugs) for block acceptance as Bitcoin Core. We have taken great care to avoid btcd causing a fork to the block chain. It includes a full block validation testing framework which contains all of the 'official' block acceptance tests (and some additional ones) that is run on every pull request to help ensure it properly follows consensus. Also, it passes all of the JSON test data in the Bitcoin Core code.

It also properly relays newly mined blocks, maintains a transaction pool, and relays individual transactions that have not yet made it into a block. It ensures all individual transactions admitted to the pool follow the rules required by the block chain and also includes more strict checks which filter transactions based on miner requirements ("standard" transactions).

One key difference between btcd and Bitcoin Core is that btcd does *NOT* include wallet functionality and this was a very intentional design decision. See the blog entry here for more details. This means you can't actually make or receive payments directly with btcd. That functionality is provided by the btcwallet and Paymetheus (Windows-only) projects which are both under active development.

CHAPTER 1

Documentation

Documentation is a work-in-progress. It is available at btcd.readthedocs.io.

CHAPTER 2

Contents

2.1 Installation

The first step is to install btcd. See one of the following sections for details on how to install on the supported operating systems.

2.1.1 Requirements

Go 1.11 or newer.

2.1.2 GPG Verification Key

All official release tags are signed by Conformal so users can ensure the code has not been tampered with and is coming from the btcsuite developers. To verify the signature perform the following:

- Download the Conformal public key: https://raw.githubusercontent.com/btcsuite/btcd/master/release/GIT-GPG-KEY-conformal.txt
- Import the public key into your GPG keyring:

```
gpg --import GIT-GPG-KEY-conformal.txt
```

- Verify the release tag with the following command where `TAG_NAME` is a placeholder for the specific tag:

```
git tag -v TAG_NAME
```

2.1.3 Windows Installation

- Install the MSI available at: btcd windows installer
- Launch btcd from the Start Menu

2.1.4 Linux/BSD/MacOSX/POSIX Installation

- Install Go according to the installation instructions
- Ensure Go was installed properly and is a supported version:

```
go version
go env GOROOT GOPATH
```

NOTE: The GOROOT and GOPATH above must not be the same path. It is recommended that GOPATH is set to a directory in your home directory such as ~/goprojects to avoid write permission issues. It is also recommended to add $GOPATH/bin to your PATH at this point.

- Run the following commands to obtain btcd, all dependencies, and install it:

```
git clone https://github.com/btcsuite/btcd $GOPATH/src/github.com/btcsuite/btcd
cd $GOPATH/src/github.com/btcsuite/btcd
GO111MODULE=on go install -v . ./cmd/...
```

- btcd (and utilities) will now be installed in $GOPATH/bin. If you did not already add the bin directory to your system path during Go installation, we recommend you do so now.

2.1.5 Gentoo Linux Installation

- Install Layman and enable the Bitcoin overlay.
- Copy or symlink /var/lib/layman/bitcoin/Documentation/package.keywords/btcd-live to /etc/portage/package.keywords/
- Install btcd: $ emerge net-p2p/btcd

2.1.6 Startup

Typically btcd will run and start downloading the block chain with no extra configuration necessary, however, there is an optional method to use a bootstrap.dat file that may speed up the initial block chain download process.

- Using bootstrap.dat

2.2 Update

- Run the following commands to update btcd, all dependencies, and install it:

```
cd $GOPATH/src/github.com/btcsuite/btcd
git pull && GO111MODULE=on go install -v . ./cmd/...
```

2.3 Configuration

btcd has a number of configuration options, which can be viewed by running: $ btcd --help.

2.3.1 Peer server listen interface

btcd allows you to bind to specific interfaces which enables you to setup configurations with varying levels of complexity. The listen parameter can be specified on the command line as shown below with the – prefix or in the configuration file without the – prefix (as can all long command line options). The configuration file takes one entry per line.

NOTE: The listen flag can be specified multiple times to listen on multiple interfaces as a couple of the examples below illustrate.

Command Line Examples:

The following config file would configure btcd to only listen on localhost for both IPv4 and IPv6:

```
[Application Options]

listen=127.0.0.1:8333
listen=[::1]:8333
```

In addition, if you are starting btcd with TLS and want to make it available via a hostname, then you will need to generate the TLS certificates for that host. For example,

```
gencerts --host=myhostname.example.com --directory=/home/me/.btcd/
```

2.3.2 RPC server listen interface

btcd allows you to bind the RPC server to specific interfaces which enables you to setup configurations with varying levels of complexity. The `rpclisten` parameter can be specified on the command line as shown below with the – prefix or in the configuration file without the – prefix (as can all long command line options). The configuration file takes one entry per line.

A few things to note regarding the RPC server:

- The RPC server will **not** be enabled unless the `rpcuser` and `rpcpass` options are specified.
- When the `rpcuser` and `rpcpass` and/or `rpclimituser` and `rpclimitpass` options are specified, the RPC server will only listen on localhost IPv4 and IPv6 interfaces by default. You will need to override the RPC listen interfaces to include external interfaces if you want to connect from a remote machine.
- The RPC server has TLS enabled by default, even for localhost. You may use the `--notls` option to disable it, but only when all listeners are on localhost interfaces.
- The `--rpclisten` flag can be specified multiple times to listen on multiple interfaces as a couple of the examples below illustrate.
- The RPC server is disabled by default when using the `--regtest` and `--simnet` networks. You can override this by specifying listen interfaces.

Command Line Examples:

The following config file would configure the btcd RPC server to listen to all interfaces on the default port, including external interfaces, for both IPv4 and IPv6:

```
[Application Options]

rpclisten=
```

2.3.3 Default ports

While btcd is highly configurable when it comes to the network configuration, the following is intended to be a quick reference for the default ports used so port forwarding can be configured as required.

btcd provides a `--upnp` flag which can be used to automatically map the bitcoin peer-to-peer listening port if your router supports UPnP. If your router does not support UPnP, or you don't wish to use it, please note that only the bitcoin peer-to-peer port should be forwarded unless you specifically want to allow RPC access to your btcd from external sources such as in more advanced network configurations.

2.3.4 Using bootstrap.dat

What is bootstrap.dat?

It is a flat, binary file containing bitcoin blockchain data starting from the genesis block and continuing through a relatively recent block height depending on the last time it was updated.

See this thread on bitcointalk for more details.

NOTE: Using bootstrap.dat is entirely optional. Btcd will download the block chain from other peers through the Bitcoin protocol with no extra configuration needed.

What are the pros and cons of using bootstrap.dat?

Pros:

- Typically accelerates the initial process of bringing up a new node as it downloads from public P2P nodes and generally is able to achieve faster download speeds
- It is particularly beneficial when bringing up multiple nodes as you only need to download the data once

Cons:

- Requires you to setup and configure a torrent client if you don't already have one available
- Requires roughly twice as much disk space since you'll need the flat file as well as the imported database

Where do I get bootstrap.dat?

The bootstrap.dat file is made available via a torrent. See this thread on bitcointalk for the torrent download details.

How do I know I can trust the bootstrap.dat I downloaded?

You don't need to trust the file as the `addblock` utility verifies every block using the same rules that are used when downloading the block chain normally through the Bitcoin protocol. Additionally, the chain rules contain hard-coded checkpoints for the known-good block chain at periodic intervals. This ensures that not only is it a valid chain, but it is the same chain that everyone else is using.

How do I use bootstrap.dat with btcd?

btcd comes with a separate utility named `addblock` which can be used to import `bootstrap.dat`. This approach is used since the import is a one-time operation and we prefer to keep the daemon itself as lightweight as possible.

1. Stop btcd if it is already running. This is required since addblock needs to access the database used by btcd and it will be locked if btcd is using it.

2. Note the path to the downloaded bootstrap.dat file.
3. Run the addblock utility with the `-i` argument pointing to the location of boostrap.dat:

Windows:

```
"%PROGRAMFILES%\Btcd Suite\Btcd\addblock" -i C:\Path\To\bootstrap.dat
```

Linux/Unix/BSD/POSIX:

```
$GOPATH/bin/addblock -i /path/to/bootstrap.dat
```

2.4 Configuring TOR

btcd provides full support for anonymous networking via the Tor Project, including *client-only* and *hidden service* configurations along with *stream isolation*. In addition, btcd supports a hybrid, *bridge mode* which is not anonymous, but allows it to operate as a bridge between regular nodes and hidden service nodes without routing the regular connections through Tor.

While it is easier to only run as a client, it is more beneficial to the Bitcoin network to run as both a client and a server so others may connect to you to as you are connecting to them. We recommend you take the time to setup a Tor hidden service for this reason.

2.4.1 Client-only

Configuring btcd as a Tor client is straightforward. The first step is obviously to install Tor and ensure it is working. Once that is done, all that typically needs to be done is to specify the `--proxy` flag via the btcd command line or in the btcd configuration file. Typically the Tor proxy address will be 127.0.0.1:9050 (if using standalone Tor) or 127.0.0.1:9150 (if using the Tor Browser Bundle). If you have Tor configured to require a username and password, you may specify them with the `--proxyuser` and `--proxypass` flags.

By default, btcd assumes the proxy specified with `--proxy` is a Tor proxy and hence will send all traffic, including DNS resolution requests, via the specified proxy.

NOTE: Specifying the `--proxy` flag disables listening by default since you will not be reachable for inbound connections unless you also configure a Tor *hidden service*.

Command line example

```
./btcd --proxy=127.0.0.1:9050
```

Config file example

```
[Application Options]

proxy=127.0.0.1:9050
```

2.4.2 Client-server via Tor hidden service

The first step is to configure Tor to provide a hidden service. Documentation for this can be found on the Tor project website here. However, there is no need to install a web server locally as the linked instructions discuss since btcd will act as the server.

In short, the instructions linked above entail modifying your `torrc` file to add something similar to the following, restarting Tor, and opening the `hostname` file in the `HiddenServiceDir` to obtain your hidden service .onion address.

```
HiddenServiceDir /var/tor/btcd
HiddenServicePort 8333 127.0.0.1:8333
```

Once Tor is configured to provide the hidden service and you have obtained your generated .onion address, configuring btcd as a Tor hidden service requires three flags:

- `--proxy` to identify the Tor (SOCKS 5) proxy to use for outgoing traffic. This is typically 127.0.0.1:9050.
- `--listen` to enable listening for inbound connections since `--proxy` disables listening by default
- `--externalip` to set the .onion address that is advertised to other peers

Command line example

```
./btcd --proxy=127.0.0.1:9050 --listen=127.0.0.1 --externalip=fooanon.onion
```

Config file example

```
[Application Options]

proxy=127.0.0.1:9050
listen=127.0.0.1
externalip=fooanon.onion
```

2.4.3 Bridge mode (not anonymous)

btcd provides support for operating as a bridge between regular nodes and hidden service nodes. In particular this means only traffic which is directed to or from a .onion address is sent through Tor while other traffic is sent normally. *As a result, this mode is **NOT** anonymous.*

This mode works by specifying an onion-specific proxy, which is pointed at Tor, by using the `--onion` flag via the btcd command line or in the btcd configuration file. If you have Tor configured to require a username and password, you may specify them with the `--onionuser` and `--onionpass` flags.

NOTE: This mode will also work in conjunction with a hidden service which means you could accept inbound connections both via the normal network and to your hidden service through the Tor network. To enable your hidden service in bridge mode, you only need to specify your hidden service's .onion address via the `--externalip` flag since traffic to and from .onion addresses are already routed via Tor due to the `--onion` flag.

Command line example

```
./btcd --onion=127.0.0.1:9050 --externalip=fooanon.onion
```

Config file example

```
[Application Options]

onion=127.0.0.1:9050
externalip=fooanon.onion
```

2.4.4 Tor stream isolation

Tor stream isolation forces Tor to build a new circuit for each connection making it harder to correlate connections.

btcd provides support for Tor stream isolation by using the `--torisolation` flag. This option requires –proxy or –onionproxy to be set.

Command line example

```
./btcd --proxy=127.0.0.1:9050 --torisolation
```

Config file example

```
[Application Options]

proxy=127.0.0.1:9050
torisolation=1
```

2.5 Using Docker

2.5.1 Introduction

With Docker you can easily set up *btcd* to run your Bitcoin full node. You can find the official *btcd* Docker images on Docker Hub btcsuite/btcd. The Docker source file of this image is located at Dockerfile.

This documentation focuses on running Docker container with *docker-compose.yml* files. These files are better to read and you can use them as a template for your own use. For more information about Docker and Docker compose visit the official Docker documentation.

2.5.2 Docker volumes

Special diskspace hint: The following examples are using a Docker managed volume. The volume is named *btcd-data* This will use a lot of disk space, because it contains the full Bitcoin blockchain. Please make yourself familiar with Docker volumes.

The *btcd-data* volume will be reused, if you upgrade your *docker-compose.yml* file. Keep in mind, that it is not automatically removed by Docker, if you delete the btcd container. If you don't need the volume anymore, please delete it manually with the command:

```
docker volume ls
docker volume rm btcd-data
```

For binding a local folder to your *btcd* container please read the Docker documentation. The preferred way is to use a Docker managed volume.

2.5.3 Known error messages when starting the btcd container

We pass all needed arguments to *btcd* as command line parameters in our *docker-compose.yml* file. It doesn't make sense to create a *btcd.conf* file. This would make things too complicated. Anyhow *btcd* will complain with following log messages when starting. These messages can be ignored:

```
Error creating a default config file: open /sample-btcd.conf: no such file or␣
↪directory
...
[WRN] BTCD: open /root/.btcd/btcd.conf: no such file or directory
```

2.5.4 Examples

Preamble

All following examples uses some defaults:

- container_name: btcd Name of the docker container that is be shown by e.g. `docker ps -a`
- hostname: btcd **(very important to set a fixed name before first start)** The internal hostname in the docker container. By default, docker is recreating the hostname every time you change the *docker-compose.yml* file. The default hostnames look like *ef00548d4fa5*. This is a problem when using the *btcd* RPC port. The RPC port is using a certificate to validate the hostname. If the hostname changes you need to recreate the certificate. To avoid this, you should set a fixed hostname before the first start. This ensures, that the docker volume is created with a certificate with this hostname.
- restart: unless-stopped Starts the *btcd* container when Docker starts, except that when the container is stopped (manually or otherwise), it is not restarted even after Docker restarts.

To use the following examples create an empty directory. In this directory create a file named *docker-compose.yml*, copy and paste the example into the *docker-compose.yml* file and run it.

```
mkdir ~/btcd-docker
cd ~/btcd-docker
touch docker-compose.yaml
nano docker-compose.yaml (use your favourite editor to edit the compose file)
docker-compose up (creates and starts a new btcd container)
```

With the following commands you can control *docker-compose*:

`docker-compose up -d` (creates and starts the container in background)

`docker-compose down` (stops and delete the container. **The docker volume btcd-data will not be deleted**)

`docker-compose stop` (stops the container)

`docker-compose start` (starts the container)

`docker ps -a` (list all running and stopped container)

`docker volume ls` (lists all docker volumes)

`docker logs btcd` (shows the log)

`docker-compose help` (brings up some helpful information)

Full node without RPC port

Let's start with an easy example. If you just want to create a full node without the need of using the RPC port, you can use the following example. This example will launch *btcd* and exposes only the default p2p port 8333 to the outside world:

```
version: "2"

services:
  btcd:
    container_name: btcd
    hostname: btcd
    image: btcsuite/btcd:latest
    restart: unless-stopped
    volumes:
      - btcd-data:/root/.btcd
    ports:
      - 8333:8333

volumes:
  btcd-data:
```

Full node with RPC port

To use the RPC port of *btcd* you need to specify a *username* and a very strong *password*. If you want to connect to the RPC port from the internet, you need to expose port 8334(RPC) as well.

```
version: "2"

services:
  btcd:
    container_name: btcd
    hostname: btcd
    image: btcsuite/btcd:latest
    restart: unless-stopped
    volumes:
      - btcd-data:/root/.btcd
    ports:
      - 8333:8333
      - 8334:8334
```

(continues on next page)

(continued from previous page)

```
    command: [
        "--rpcuser=[CHOOSE_A_USERNAME]",
        "--rpcpass=[CREATE_A_VERY_HARD_PASSWORD]"
    ]

volumes:
  btcd-data:
```

Full node with RPC port running on TESTNET

To run a node on testnet, you need to provide the *–testnet* argument. The ports for testnet are 18333 (p2p) and 18334 (RPC):

```
version: "2"

services:
  btcd:
    container_name: btcd
    hostname: btcd
    image: btcsuite/btcd:latest
    restart: unless-stopped
    volumes:
      - btcd-data:/root/.btcd
    ports:
      - 18333:18333
      - 18334:18334
    command: [
        "--testnet",
        "--rpcuser=[CHOOSE_A_USERNAME]",
        "--rpcpass=[CREATE_A_VERY_HARD_PASSWORD]"
    ]

volumes:
  btcd-data:
```

2.6 Controlling and querying btcd via btcctl

btcctl is a command line utility that can be used to both control and query btcd via RPC. btcd does **not** enable its RPC server by default; You must configure at minimum both an RPC username and password or both an RPC limited username and password:

- btcd.conf configuration file

```
[Application Options]
rpcuser=myuser
rpcpass=SomeDecentp4ssw0rd
rpclimituser=mylimituser
rpclimitpass=Limitedp4ssw0rd
```

- btcctl.conf configuration file

```
[Application Options]
rpcuser=myuser
rpcpass=SomeDecentp4ssw0rd
```

OR

```
[Application Options]
rpclimituser=mylimituser
rpclimitpass=Limitedp4ssw0rd
```

For a list of available options, run: `$ btcctl --help`

2.7 Mining

btcd supports the `getblocktemplate` RPC. The limited user cannot access this RPC.

2.7.1 Add the payment addresses with the `miningaddr` option

```
[Application Options]
rpcuser=myuser
rpcpass=SomeDecentp4ssw0rd
miningaddr=12c6DSiU4Rq3P4ZxziKxzrL5LmMBrzjrJX
miningaddr=1M83ju3EChKYyysmM2FXtLNftbacagd8FR
```

2.7.2 Add btcd's RPC TLS certificate to system Certificate Authority list

`cgminer` uses curl to fetch data from the RPC server. Since curl validates the certificate by default, we must install the `btcd` RPC certificate into the default system Certificate Authority list.

2.7.3 Ubuntu

1. Copy rpc.cert to /usr/share/ca-certificates: `# cp /home/user/.btcd/rpc.cert /usr/share/ca-certificates/btcd.crt`
2. Add btcd.crt to /etc/ca-certificates.conf: `# echo btcd.crt >> /etc/ca-certificates.conf`
3. Update the CA certificate list: `# update-ca-certificates`

2.7.4 Set your mining software url to use https

```
cgminer -o https://127.0.0.1:8334 -u rpcuser -p rpcpassword
```

2.8 Wallet

btcd was intentionally developed without an integrated wallet for security reasons. Please see btcwallet for more information.

2.9 Developer Resources

- Code Contribution Guidelines
- JSON-RPC Reference
  - RPC Examples
- The btcsuite Bitcoin-related Go Packages:
  - btcrpcclient - Implements a robust and easy to use Websocket-enabled Bitcoin JSON-RPC client
  - btcjson - Provides an extensive API for the underlying JSON-RPC command and return values
  - wire - Implements the Bitcoin wire protocol
  - peer - Provides a common base for creating and managing Bitcoin network peers.
  - blockchain - Implements Bitcoin block handling and chain selection rules
  - blockchain/fullblocktests - Provides a set of block tests for testing the consensus validation rules
  - txscript - Implements the Bitcoin transaction scripting language
  - btcec - Implements support for the elliptic curve cryptographic functions needed for the Bitcoin scripts
  - database - Provides a database interface for the Bitcoin block chain
  - mempool - Package mempool provides a policy-enforced pool of unmined bitcoin transactions.
  - btcutil - Provides Bitcoin-specific convenience functions and types
  - chainhash - Provides a generic hash type and associated functions that allows the specific hash algorithm to be abstracted.
  - connmgr - Package connmgr implements a generic Bitcoin network connection manager.

2.10 JSON RPC API

2.10.1 1. Overview

btcd provides a JSON-RPC API that is fully compatible with the original bitcoind/bitcoin-qt. There are a few key differences between btcd and bitcoind as far as how RPCs are serviced:

- Unlike bitcoind that has the wallet and chain intermingled in the same process which leads to several issues, btcd intentionally splits the wallet and chain services into independent processes. See the blog post here for further details on why they were separated. This means that if you are talking directly to btcd, only chain-related RPCs are available. However both chain-related and wallet-related RPCs are available via btcwallet.
- btcd is secure by default which means that the RPC connection is TLS-enabled by default
- btcd provides access to the API through both HTTP POST requests and Websockets

Websockets are the preferred transport for btcd RPC and are used by applications such as btcwallet for inter-process communication with btcd. The websocket connection endpoint for btcd is `wss://your_ip_or_domain:8334/ws`.

In addition to the *standard API*, an *extension API* has been developed that is exclusive to clients using Websockets. In its current state, this API attempts to cover features found missing in the standard API during the development of btcwallet.

While the *standard API* is stable, the *Websocket extension API* should be considered a work in progress, incomplete, and susceptible to changes (both additions and removals).

The original bitcoind/bitcoin-qt JSON-RPC API documentation is available at https://en.bitcoin.it/wiki/Original_Bitcoin_client/API_Calls_list

2.10.2 2. HTTP POST Versus Websockets

The btcd RPC server supports both HTTP POST requests and the preferred Websockets. All of the *standard* and *extension* methods described in this documentation can be accessed through both. As the name indicates, the *Websocket-specific extension* methods can only be accessed when connected via Websockets.

As mentioned in the *overview*, the websocket connection endpoint for btcd is `wss://your_ip_or_domain:8334/ws`.

The most important differences between the two transports as it pertains to the JSON-RPC API are:

2.10.3 3. Authentication

3.1 Authentication Overview

The following authentication details are needed before establishing a connection to a btcd RPC server:

- **rpcuser** is the full-access username configured for the btcd RPC server
- **rpcpass** is the full-access password configured for the btcd RPC server
- **rpclimituser** is the limited username configured for the btcd RPC server
- **rpclimitpass** is the limited password configured for the btcd RPC server
- **rpccert** is the PEM-encoded X.509 certificate (public key) that the btcd server is configured with. It is automatically generated by btcd and placed in the btcd home directory (which is typically `%LOCALAPPDATA%\Btcd` on Windows and `~/.btcd` on POSIX-like OSes)

NOTE: As mentioned above, btcd is secure by default which means the RPC server is not running unless configured with a **rpcuser** and **rpcpass** and/or a **rpclimituser** and **rpclimitpass**, and uses TLS authentication for all connections.

Depending on which connection transaction you are using, you can choose one of two, mutually exclusive, methods.

- *Use HTTP Authorization Header* - HTTP POST requests and Websockets
- *Use the JSON-RPC "authenticate" command* - Websockets only

3.2 HTTP Basic Access Authentication

The btcd RPC server uses HTTP basic access authentication with the **rpcuser** and **rpcpass** detailed above. If the supplied credentials are invalid, you will be disconnected immediately upon making the connection.

3.3 JSON-RPC Authenticate Command (Websocket-specific)

While the HTTP basic access authentication method is the preferred method, the ability to set HTTP headers from websockets is not always available. In that case, you will need to use the *authenticate* JSON-RPC method.

The *authenticate* command must be the first command sent after connecting to the websocket. Sending any other commands before authenticating, supplying invalid credentials, or attempting to authenticate again when already authenticated will cause the websocket to be closed immediately.

2.10.4 4. Command-line Utility

btcd comes with a separate utility named `btcctl` which can be used to issue these RPC commands via HTTP POST requests to btcd after configuring it with the information in the *Authentication* section above. It can also be used to communicate with any server/daemon/service which provides a JSON-RPC API compatible with the original bitcoind/bitcoin-qt client.

2.10.5 5. Standard Methods

5.1 Method Overview

The following is an overview of the RPC methods and their current status. Click the method name for further details such as parameter and return information.

5.2 Method Details

2.10.6 6. Extension Methods

6.1 Method Overview

The following is an overview of the RPC methods which are implemented by btcd, but not the original bitcoind client. Click the method name for further details such as parameter and return information.

6.2 Method Details

2.10.7 7. Websocket Extension Methods (Websocket-specific)

7.1 Method Overview

The following is an overview of the RPC method requests available exclusively to Websocket clients. All of these RPC methods are available to the limited user. Click the method name for further details such as parameter and return information.

7.2 Method Details

2.10.8 8. Notifications (Websocket-specific)

btcd uses standard JSON-RPC notifications to notify clients of changes, rather than requiring clients to poll btcd for updates. JSON-RPC notifications are a subset of requests, but do not contain an ID. The notification type is categorized by the `method` field and additional details are sent as a JSON array in the `params` field.

8.1 Notification Overview

The following is an overview of the JSON-RPC notifications used for Websocket connections. Click the method name for further details of the context(s) in which they are sent and their parameters.

8.2 Notification Details

2.10.9 9. Example Code

This section provides example code for interacting with the JSON-RPC API in various languages.

- *Go*
- *node.js*

9.1 Go

This section provides examples of using the RPC interface using Go and the rpcclient package.

- *Using getblockcount to Retrieve the Current Block Height*
- *Using getblock to Retrieve the Genesis Block*
- *Using notifyblocks to Receive blockconnected and blockdisconnected Notifications (Websocket-specific)*

9.1.1 Using getblockcount to Retrieve the Current Block Height

The following is an example Go application which uses the rpcclient package to connect with a btcd instance via Websockets, issues *getblockcount* to retrieve the current block height, and displays it.

```go
package main

import (
        "io/ioutil"
        "log"
        "path/filepath"

        "github.com/btcsuite/btcd/rpcclient"
        "github.com/btcsuite/btcutil"
)

func main() {
        // Load the certificate for the TLS connection which is automatically
        // generated by btcd when it starts the RPC server and doesn't already
        // have one.
        btcdHomeDir := btcutil.AppDataDir("btcd", false)
        certs, err := ioutil.ReadFile(filepath.Join(btcdHomeDir, "rpc.cert"))
        if err != nil {
                log.Fatal(err)
        }

        // Create a new RPC client using websockets.  Since this example is
        // not long-lived, the connection will be closed as soon as the program
        // exits.
        connCfg := &rpcclient.ConnConfig{
                Host:         "localhost:8334",
                Endpoint:     "ws",
                User:         "yourrpcuser",
                Pass:         "yourrpcpass",
                Certificates: certs,
        }
        client, err := rpcclient.New(connCfg, nil)
        if err != nil {
                log.Fatal(err)
        }
        defer client.Shutdown()

        // Query the RPC server for the current block count and display it.
```

(continues on next page)

(continued from previous page)

```
        blockCount, err := client.GetBlockCount()
        if err != nil {
                log.Fatal(err)
        }
        log.Printf("Block count: %d", blockCount)
}
```

Which results in:

```
2018/08/27 11:17:27 Block count: 536027
```

9.1.2 Using getblock to Retrieve the Genesis Block

The following is an example Go application which uses the rpcclient package to connect with a btcd instance via Websockets, issues *getblock* to retrieve information about the Genesis block, and display a few details about it.

```
package main

import (
        "io/ioutil"
        "log"
        "path/filepath"
        "time"

        "github.com/btcsuite/btcd/chaincfg/chainhash"
        "github.com/btcsuite/btcd/rpcclient"
        "github.com/btcsuite/btcutil"
)

func main() {
        // Load the certificate for the TLS connection which is automatically
        // generated by btcd when it starts the RPC server and doesn't already
        // have one.
        btcdHomeDir := btcutil.AppDataDir("btcd", false)
        certs, err := ioutil.ReadFile(filepath.Join(btcdHomeDir, "rpc.cert"))
        if err != nil {
                log.Fatal(err)
        }

        // Create a new RPC client using websockets.  Since this example is
        // not long-lived, the connection will be closed as soon as the program
        // exits.
        connCfg := &rpcclient.ConnConfig{
                Host:         "localhost:18334",
                Endpoint:     "ws",
                User:         "yourrpcuser",
                Pass:         "yourrpcpass",
                Certificates: certs,
        }
        client, err := rpcclient.New(connCfg, nil)
        if err != nil {
                log.Fatal(err)
        }
        defer client.Shutdown()

        // Query the RPC server for the genesis block using the "getblock"
        // command with the verbose flag set to true and the verboseTx flag
```

(continues on next page)

(continued from previous page)

```
        // set to false.
        genesisHashStr :=
↪"000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f"
        blockHash, err := chainhash.NewHashFromStr(genesisHashStr)
        if err != nil {
                log.Fatal(err)
        }
        block, err := client.GetBlockVerbose(blockHash)
        if err != nil {
                log.Fatal(err)
        }

        // Display some details about the returned block.
        log.Printf("Hash: %v\n", block.Hash)
        log.Printf("Previous Block: %v\n", block.PreviousHash)
        log.Printf("Next Block: %v\n", block.NextHash)
        log.Printf("Merkle root: %v\n", block.MerkleRoot)
        log.Printf("Timestamp: %v\n", time.Unix(block.Time, 0).UTC())
        log.Printf("Confirmations: %v\n", block.Confirmations)
        log.Printf("Difficulty: %f\n", block.Difficulty)
        log.Printf("Size (in bytes): %v\n", block.Size)
        log.Printf("Num transactions: %v\n", len(block.Tx))
}
```

Which results in:

```
Hash: 000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f
Previous Block: 0000000000000000000000000000000000000000000000000000000000000000
Next Block: 00000000839a8e6886ab5951d76f411475428afc90947ee320161bbf18eb6048
Merkle root: 4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b
Timestamp: 2009-01-03 18:15:05 +0000 UTC
Confirmations: 534323
Difficulty: 1.000000
Size (in bytes): 285
Num transactions: 1
```

9.1.3 Using notifyblocks to Receive blockconnected and blockdisconnected Notifications (Websocket-specific)

The following is an example Go application which uses the rpcclient package to connect with a btcd instance via Websockets and registers for *blockconnected* and *blockdisconnected* notifications with *notifyblocks*. It also sets up handlers for the notifications.

```
package main

import (
        "io/ioutil"
        "log"
        "path/filepath"
        "time"

        "github.com/btcsuite/btcd/chaincfg/chainhash"
        "github.com/btcsuite/btcd/rpcclient"
        "github.com/btcsuite/btcutil"
)

func main() {
        // Setup handlers for blockconnected and blockdisconnected
```

(continues on next page)

(continued from previous page)

```
        // notifications.
        ntfnHandlers := rpcclient.NotificationHandlers{
                OnBlockConnected: func(hash *chainhash.Hash, height int32, t time.
↪Time) {
                        log.Printf("Block connected: %v (%d) %s", hash, height, t)
                },
                OnBlockDisconnected: func(hash *chainhash.Hash, height int32, t time.
↪Time) {
                        log.Printf("Block disconnected: %v (%d) %s", hash, height, t)
                },
        }

        // Load the certificate for the TLS connection which is automatically
        // generated by btcd when it starts the RPC server and doesn't already
        // have one.
        btcdHomeDir := btcutil.AppDataDir("btcd", false)
        certs, err := ioutil.ReadFile(filepath.Join(btcdHomeDir, "rpc.cert"))
        if err != nil {
                log.Fatal(err)
        }

        // Create a new RPC client using websockets.
        connCfg := &rpcclient.ConnConfig{
                Host:         "localhost:8334",
                Endpoint:     "ws",
                User:         "yourrpcuser",
                Pass:         "yourrpcpass",
                Certificates: certs,
        }
        client, err := rpcclient.New(connCfg, &ntfnHandlers)
        if err != nil {
                log.Fatal(err)
        }

        // Register for blockconnected and blockdisconneted notifications.
        if err := client.NotifyBlocks(); err != nil {
                client.Shutdown()
                log.Fatal(err)
        }

        // For this example, gracefully shutdown the client after 10 seconds.
        // Ordinarily when to shutdown the client is highly application
        // specific.
        log.Println("Client shutdown in 10 seconds...")
        time.AfterFunc(time.Second*10, func() {
                log.Println("Client shutting down...")
                client.Shutdown()
                log.Println("Client shutdown complete.")
        })

        // Wait until the client either shuts down gracefully (or the user
        // terminates the process with Ctrl+C).
        client.WaitForShutdown()
}
```

Example output:

```
2018/08/27 10:35:43 Client shutdown in 10 seconds...
2018/08/27 10:35:44 Block connected:␣
↪00000000000000000003321723557df58914658dc6fd963d547292a0a4797454 (534747) 2018-08-
↪02 06:37:52 +0800 CST
2018/08/27 10:35:47 Block connected:␣
↪0000000000000000002e12773b798fc61dffe00ed5c3e89d3c306f8058c51e13 (534748) 2018-08-
↪02 06:39:54 +0800 CST
2018/08/27 10:35:49 Block connected:␣
↪0000000000000000001bb311cd849839ce88499b91a201922f55a1cfafabe267 (534749) 2018-08-
↪02 06:44:22 +0800 CST
2018/08/27 10:35:50 Block connected:␣
↪00000000000000000019d7296c9b5c175369ad337ec44b76bd4728021a09b864 (534750) 2018-08-
↪02 06:55:44 +0800 CST
2018/08/27 10:35:53 Block connected:␣
↪00000000000000000022db98cf47e944ed58ca450c819e8fef8f8c71ca5d9901 (534751) 2018-08-
↪02 06:57:39 +0800 CST
2018/08/27 10:35:53 Client shutting down...
2018/08/27 10:35:53 Client shutdown complete.
```

2.10.10 9.2. Example node.js Code

9.2.1 Using notifyblocks to be Notified of Block Connects and Disconnects

The following is example node.js code which uses ws (can be installed with `npm install ws`) to connect with a btcd instance, issues *notifyblocks* to register for *blockconnected* and *blockdisconnected* notifications, and displays all incoming messages.

```
var fs = require('fs');
var WebSocket = require('ws');

// Load the certificate for the TLS connection which is automatically
// generated by btcd when it starts the RPC server and doesn't already
// have one.
var cert = fs.readFileSync('/path/to/btcd/appdata/rpc.cert');
var user = "yourusername";
var password = "yourpassword";


// Initiate the websocket connection.  The btcd generated certificate acts as
// its own certificate authority, so it needs to be specified in the 'ca' array
// for the certificate to properly validate.
var ws = new WebSocket('wss://127.0.0.1:8334/ws', {
  headers: {
    'Authorization': 'Basic '+new Buffer(user+':'+password).toString('base64')
  },
  cert: cert,
  ca: [cert]
});
ws.on('open', function() {
    console.log('CONNECTED');
    // Send a JSON-RPC command to be notified when blocks are connected and
    // disconnected from the chain.
    ws.send('{"jsonrpc":"1.0","id":"0","method":"notifyblocks","params":[]}');
});
ws.on('message', function(data, flags) {
    console.log(data);
```

(continues on next page)

(continued from previous page)

```
});
ws.on('error', function(derp) {
  console.log('ERROR:' + derp);
})
ws.on('close', function(data) {
  console.log('DISCONNECTED');
})
```

2.11 Code contribution guidelines

Developing cryptocurrencies is an exciting endeavor that touches a wide variety of areas such as wire protocols, peer-to-peer networking, databases, cryptography, language interpretation (transaction scripts), RPC, and websockets. They also represent a radical shift to the current fiscal system and as a result provide an opportunity to help reshape the entire financial system. There are few projects that offer this level of diversity and impact all in one code base.

However, as exciting as it is, one must keep in mind that cryptocurrencies represent real money and introducing bugs and security vulnerabilities can have far more dire consequences than in typical projects where having a small bug is minimal by comparison. In the world of cryptocurrencies, even the smallest bug in the wrong area can cost people a significant amount of money. For this reason, the btcd suite has a formalized and rigorous development process which is outlined on this page.

We highly encourage code contributions, however it is imperative that you adhere to the guidelines established on this page.

2.11.1 Minimum Recommended Skillset

The following list is a set of core competencies that we recommend you possess before you really start attempting to contribute code to the project. These are not hard requirements as we will gladly accept code contributions as long as they follow the guidelines set forth on this page. That said, if you don't have the following basic qualifications you will likely find it quite difficult to contribute.

- A reasonable understanding of bitcoin at a high level (see the *Required Reading* section for the original white paper)
- Experience in some type of C-like language
- An understanding of data structures and their performance implications
- Familiarity with unit testing
- Debugging experience
- Ability to understand not only the area you are making a change in, but also the code your change relies on, and the code which relies on your changed code

Building on top of those core competencies, the recommended skill set largely depends on the specific areas you are looking to contribute to. For example, if you wish to contribute to the cryptography code, you should have a good understanding of the various aspects involved with cryptography such as the security and performance implications.

2.11.2 Required Reading

- Effective Go - The entire btcd suite follows the guidelines in this document. For your code to be accepted, it must follow the guidelines therein.

- Original Satoshi Whitepaper - This is the white paper that started it all. Having a solid foundation to build on will make the code much more comprehensible.

2.11.3 Development Practices

Developers are expected to work in their own trees and submit pull requests when they feel their feature or bug fix is ready for integration into the master branch.

2.11.4 Share Early, Share Often

We firmly believe in the share early, share often approach. The basic premise of the approach is to announce your plans **before** you start work, and once you have started working, craft your changes into a stream of small and easily reviewable commits.

This approach has several benefits:

- Announcing your plans to work on a feature **before** you begin work avoids duplicate work
- It permits discussions which can help you achieve your goals in a way that is consistent with the existing architecture
- It minimizes the chances of you spending time and energy on a change that might not fit with the consensus of the community or existing architecture and potentially be rejected as a result
- Incremental development helps ensure you are on the right track with regards to the rest of the community
- The quicker your changes are merged to master, the less time you will need to spend rebasing and otherwise trying to keep up with the main code base

2.11.5 Testing

One of the major design goals of all core btcd packages is to aim for complete test coverage. This is financial software so bugs and regressions can cost people real money. For this reason every effort must be taken to ensure the code is as accurate and bug-free as possible. Thorough testing is a good way to help achieve that goal.

Unless a new feature you submit is completely trivial, it will probably be rejected unless it is also accompanied by adequate test coverage for both positive and negative conditions. That is to say, the tests must ensure your code works correctly when it is fed correct data as well as incorrect data (error paths).

Go provides an excellent test framework that makes writing test code and checking coverage statistics straight forward. For more information about the test coverage tools, see the golang cover blog post.

A quick summary of test practices follows:

- All new code should be accompanied by tests that ensure the code behaves correctly when given expected values, and, perhaps even more importantly, that it handles errors gracefully
- When you fix a bug, it should be accompanied by tests which exercise the bug to both prove it has been resolved and to prevent future regressions

2.11.6 Code Documentation and Commenting

- At a minimum every function must be commented with its intended purpose and any assumptions that it makes
  - Function comments must always begin with the name of the function per Effective Go

  – Function comments should be complete sentences since they allow a wide variety of automated presentations such as godoc.org
  – The general rule of thumb is to look at it as if you were completely unfamiliar with the code and ask yourself, would this give me enough information to understand what this function does and how I'd probably want to use it?

- Exported functions should also include detailed information the caller of the function will likely need to know and/or understand:

WRONG

```
// convert a compact uint32 to big.Int
func CompactToBig(compact uint32) *big.Int {
```

RIGHT

```
// CompactToBig converts a compact representation of a whole number N to a
// big integer.  The representation is similar to IEEE754 floating point
// numbers.
//
// Like IEEE754 floating point, there are three basic components: the sign,
// the exponent, and the mantissa. They are broken out as follows:
//
//        * the most significant 8 bits represent the unsigned base 256 exponent
//        * bit 23 (the 24th bit) represents the sign bit
//        * the least significant 23 bits represent the mantissa
//
//        -------------------------------------------------
//        |   Exponent     |    Sign    |    Mantissa     |
//        -------------------------------------------------
//        | 8 bits [31-24] | 1 bit [23] | 23 bits [22-00] |
//        -------------------------------------------------
//
// The formula to calculate N is:
//         N = (-1^sign) * mantissa * 256^(exponent-3)
//
// This compact form is only used in bitcoin to encode unsigned 256-bit numbers
// which represent difficulty targets, thus there really is not a need for a
// sign bit, but it is implemented here to stay consistent with bitcoind.
func CompactToBig(compact uint32) *big.Int {
```

- Comments in the body of the code are highly encouraged, but they should explain the intention of the code as opposed to just calling out the obvious

WRONG

```
// return err if amt is less than 5460
if amt < 5460 {
  return err
}
```

RIGHT

```
// Treat transactions with amounts less than the amount which is considered dust
// as non-standard.
if amt < 5460 {
  return err
}
```

NOTE: The above should really use a constant as opposed to a magic number, but it was left as a magic number to show how much of a difference a good comment can make.

2.11.7 Model Git Commit Messages

This project prefers to keep a clean commit history with well-formed commit messages. This section illustrates a model commit message and provides a bit of background for it. This content was originally created by Tim Pope and made available on his website, however that website is no longer active, so it is being provided here.

Here's a model Git commit message:

```
Short (50 chars or less) summary of changes

More detailed explanatory text, if necessary.  Wrap it to about 72
characters or so.  In some contexts, the first line is treated as the
subject of an email and the rest of the text as the body.  The blank
line separating the summary from the body is critical (unless you omit
the body entirely); tools like rebase can get confused if you run the
two together.

Write your commit message in the present tense: "Fix bug" and not "Fixed
bug."  This convention matches up with commit messages generated by
commands like git merge and git revert.

Further paragraphs come after blank lines.

- Bullet points are okay, too
- Typically a hyphen or asterisk is used for the bullet, preceded by a
  single space, with blank lines in between, but conventions vary here
- Use a hanging indent
```

Prefix the summary with the subsystem/package when possible. Many other projects make use of the code and this makes it easier for them to tell when something they're using has changed. Have a look at past commits for examples of commit messages.

Here are some of the reasons why wrapping your commit messages to 72 columns is a good thing.

- git log doesn't do any special special wrapping of the commit messages. With the default pager of less -S, this means your paragraphs flow far off the edge of the screen, making them difficult to read. On an 80 column terminal, if we subtract 4 columns for the indent on the left and 4 more for symmetry on the right, we're left with 72 columns.
- git format-patch –stdout converts a series of commits to a series of emails, using the messages for the message body. Good email netiquette dictates we wrap our plain text emails such that there's room for a few levels of nested reply indicators without overflow in an 80 column terminal.

2.11.8 Code Approval Process

This section describes the code approval process that is used for code contributions. This is how to get your changes into btcd.

2.11.9 Code Review

All code which is submitted will need to be reviewed before inclusion into the master branch. This process is performed by the project maintainers and usually other committers who are interested in the area you are working in as well.

2.11.10 Code Review Timeframe

The timeframe for a code review will vary greatly depending on factors such as the number of other pull requests which need to be reviewed, the size and complexity of the contribution, how well you followed the guidelines presented on this page, and how easy it is for the reviewers to digest your commits. For example, if you make one monolithic commit that makes sweeping changes to things in multiple subsystems, it will obviously take much longer to review. You will also likely be asked to split the commit into several smaller, and hence more manageable, commits.

Keeping the above in mind, most small changes will be reviewed within a few days, while large or far reaching changes may take weeks. This is a good reason to stick with the *Share Early, Share Often* development practice outlined above.

2.11.11 What is the review looking for?

The review is mainly ensuring the code follows the *Development Practices* and *Code Contribution Standards*. However, there are a few other checks which are generally performed as follows:

- The code is stable and has no stability or security concerns
- The code is properly using existing APIs and generally fits well into the overall architecture
- The change is not something which is deemed inappropriate by community consensus

2.11.12 Rework Code (if needed)

After the code review, the change will be accepted immediately if no issues are found. If there are any concerns or questions, you will be provided with feedback along with the next steps needed to get your contribution merged with master. In certain cases the code reviewer(s) or interested committers may help you rework the code, but generally you will simply be given feedback for you to make the necessary changes.

This process will continue until the code is finally accepted.

2.11.13 Acceptance

Once your code is accepted, it will be integrated with the master branch. Typically it will be rebased and fast-forward merged to master as we prefer to keep a clean commit history over a tangled weave of merge commits. However, regardless of the specific merge method used, the code will be integrated with the master branch and the pull request will be closed.

Rejoice as you will now be listed as a contributor!

2.11.14 Contribution Standards

2.11.15 Contribution Checklist

- [ ] All changes are Go version 1.3 compliant
- [ ] The code being submitted is commented according to the *Code Documentation and Commenting* section
- [ ] For new code: Code is accompanied by tests which exercise both the positive and negative (error paths) conditions (if applicable)
- [ ] For bug fixes: Code is accompanied by new tests which trigger the bug being fixed to prevent regressions
- [ ] Any new logging statements use an appropriate subsystem and logging level
- [ ] Code has been formatted with `go fmt`

- [ ] Running `go test` does not fail any tests
- [ ] Running `go vet` does not report any issues
- [ ] Running golint does not report any **new** issues that did not already exist

2.11.16 Licensing of Contributions

All contributions must be licensed with the ISC license. This is the same license as all of the code in the btcd suite.

2.12 Contact

2.12.1 IRC

- irc.freenode.net, channel `#btcd`

2.12.2 Mailing Lists

- btcd: discussion of btcd and its packages.
- btcd-commits: readonly mail-out of source code changes.

2.12.3 Issue Tracker

The integrated github issue tracker is used for this project.

CHAPTER 3

License

btcd is licensed under the copyfree ISC License.