Bitcoin Core: The Reference Implementation

Bitcoin is an *open source* project and the source code is available under an open (MIT) license, free to download and use for any purpose. Open source means more than simply free to use. It also means that bitcoin is developed by an open community of volunteers. At first, that community consisted of only Satoshi Nakamoto. By 2016, bitcoin's source code had more than 400 contributors with about a dozen developers working on the code almost full-time and several dozen more on a part-time basis. Anyone can contribute to the code—including you!

When bitcoin was created by Satoshi Nakamoto, the software was actually completed before the whitepaper reproduced in [satoshi_whitepaper] was written. Satoshi wanted to make sure it worked before writing about it. That first implementation, then simply known as "Bitcoin" or "Satoshi client," has been heavily modified and improved. It has evolved into what is known as *Bitcoin Core*, to differentiate it from other compatible implementations. Bitcoin Core is the *reference implementation* of the bitcoin system, meaning that it is the authoritative reference on how each part of the technology should be implemented. Bitcoin Core implements all aspects of bitcoin, including wallets, a transaction and block validation engine, and a full network node in the peer-to-peer bitcoin network.

WARNING	Even though Bitcoin Core includes a reference implementation of a wallet, this is not intended to be used as a production wallet for users or for applications. Application developers are advised to build wallets using modern standards such as BIP-39 and BIP-32 (see [mnemonic_code_words] and [hd_wallets]). BIP stands for *Bitcoin Improvement Proposal*.

Bitcoin Core architecture (Source: Eric Lombrozo) shows the architecture of Bitcoin Core.

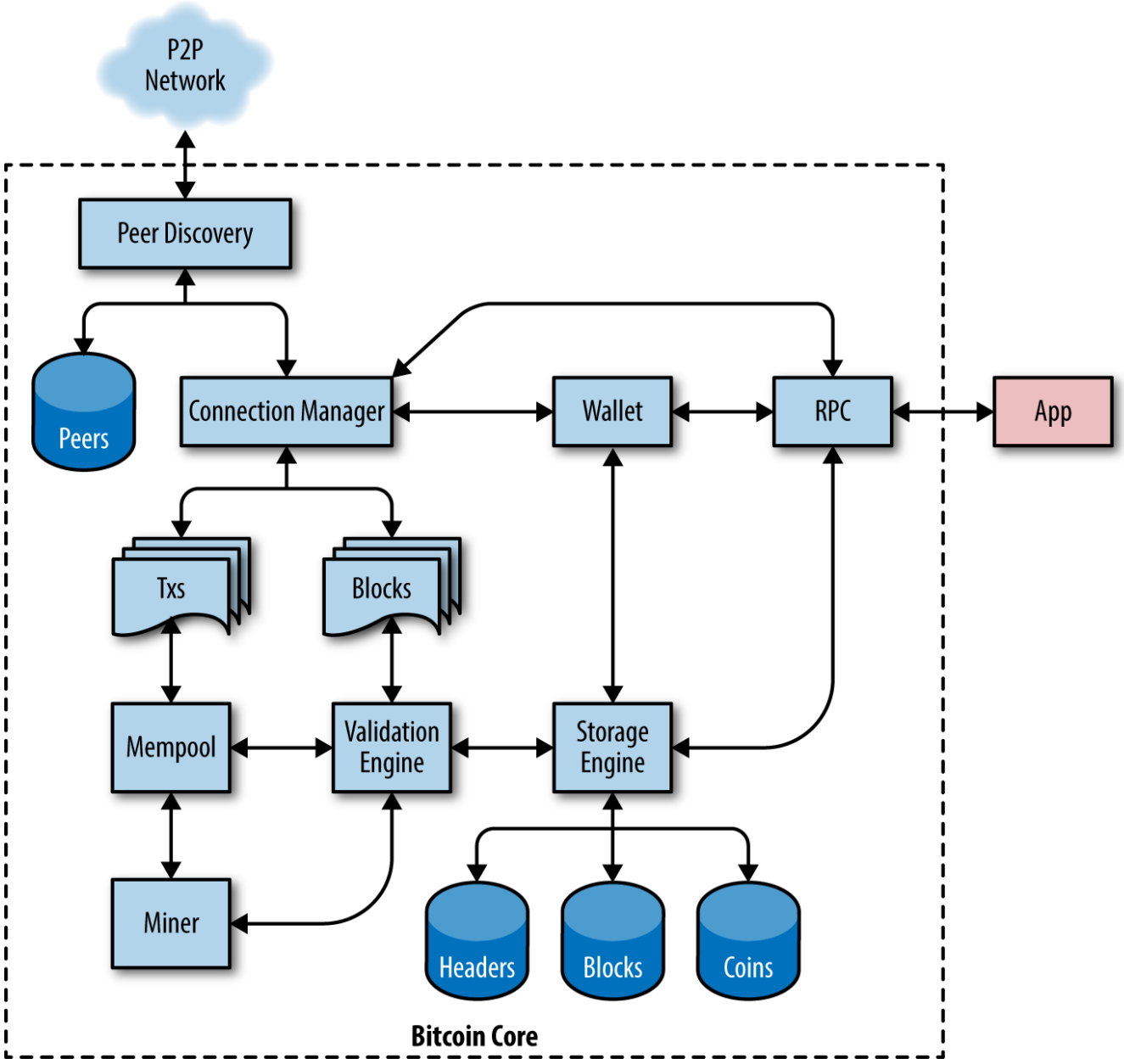

Figure 1. Bitcoin Core architecture (Source: Eric Lombrozo)

Bitcoin Development Environment

If you're a developer, you will want to set up a development environment with all the tools, libraries, and support software for writing bitcoin applications. In this highly technical chapter, we'll walk through that process step-by-step. If the material becomes too dense (and you're not actually setting up a development environment) feel free to skip to the next chapter, which is less technical.

Compiling Bitcoin Core from the Source Code

Bitcoin Core's source code can be downloaded as an archive or by cloning the authoritative source repository from GitHub. On the Bitcoin Core download page, select the most recent version and download the compressed archive of the source code, e.g., bitcoin-0.15.0.2.tar.gz. Alternatively, use the git command line to create a local copy of the source code from the GitHub bitcoin page.

TIP

In many of the examples in this chapter we will be using the operating system's command-line interface (also known as a "shell"), accessed via a "terminal" application. The shell will display a prompt; you type a command; and the shell responds with some text and a new prompt for your next command. The prompt may look different on your system, but in the following examples it is denoted by a $ symbol. In the examples, when you see text after a $ symbol, don't type the $ symbol but type the command immediately following it, then press Enter to execute the command. In the examples, the lines below each command are the operating system's responses to that command. When you see the next $ prefix, you'll know it's a new command and you should repeat the process.

In this example, we are using the git command to create a local copy ("clone") of the source code:

```
$ git clone https://github.com/bitcoin/bitcoin.git
Cloning into 'bitcoin'...
remote: Counting objects: 102071, done.
remote: Compressing objects: 100% (10/10), done.
Receiving objects: 100% (102071/102071), 86.38 MiB | 730.00 KiB/s, done.
remote: Total 102071 (delta 4), reused 5 (delta 1), pack-reused 102060
Resolving deltas: 100% (76168/76168), done.
Checking connectivity... done.
$
```

TIP

Git is the most widely used distributed version control system, an essential part of any software developer's toolkit. You may need to install the git command, or a graphical user interface for git, on your operating system if you do not have it already.

When the git cloning operation has completed, you will have a complete local copy of the source code repository in the directory *bitcoin*. Change to this directory by typing **cd bitcoin** at the prompt:

```
$ cd bitcoin
```

Selecting a Bitcoin Core Release

By default, the local copy will be synchronized with the most recent code, which might be an unstable or beta version of bitcoin. Before compiling the code, select a specific version by checking out a release *tag*. This will synchronize the local copy with a specific snapshot of the code repository identified by a keyword tag. Tags are used by the developers to mark specific releases of the code by version number. First, to find the available tags, we use the git tag command:

```
$ git tag
v0.1.5
v0.1.6test1
v0.10.0
...
v0.11.2
v0.11.2rc1
v0.12.0rc1
v0.12.0rc2
...
```

The list of tags shows all the released versions of bitcoin. By convention, *release candidates*, which are intended for testing, have the suffix "rc." Stable releases that can be run on production systems have no suffix. From the preceding list, select the highest version release, which at the time of writing was v0.15.0. To synchronize the local code with this version, use the git checkout command:

```
$ git checkout v0.15.0
HEAD is now at 3751912... Merge #11295: doc: Old fee_estimates.dat are discarded by
0.15.0
```

You can confirm you have the desired version "checked out" by issuing the command git status:

```
$ git status
HEAD detached at v0.15.0
nothing to commit, working directory clean
```

Configuring the Bitcoin Core Build

The source code includes documentation, which can be found in a number of files. Review the main documentation located in *README.md* in the *bitcoin* directory by typing **more README.md** at the prompt and using the spacebar to progress to the next page. In this chapter, we will build the command-line bitcoin client, also known as bitcoind on Linux. Review the instructions for compiling the bitcoind command-line client on your platform by typing **more doc/build-unix.md**. Alternative instructions for macOS and Windows can be found in the *doc* directory, as *build-osx.md* or *build-windows.md*, respectively.

Carefully review the build prerequisites, which are in the first part of the build documentation. These are libraries that must be present on your system before you can begin to compile bitcoin. If these prerequisites are missing, the build process will fail with an error. If this happens because you missed a prerequisite, you can install it and then resume the build process from where you left off. Assuming the prerequisites are installed, you start the build process by generating a set of build scripts using the *autogen.sh* script.

```
$ ./autogen.sh
...
glibtoolize: copying file 'build-aux/m4/libtool.m4'
glibtoolize: copying file 'build-aux/m4/ltoptions.m4'
glibtoolize: copying file 'build-aux/m4/ltsugar.m4'
glibtoolize: copying file 'build-aux/m4/ltversion.m4'
...
configure.ac:10: installing 'build-aux/compile'
configure.ac:5: installing 'build-aux/config.guess'
configure.ac:5: installing 'build-aux/config.sub'
configure.ac:9: installing 'build-aux/install-sh'
configure.ac:9: installing 'build-aux/missing'
Makefile.am: installing 'build-aux/depcomp'
...
```

The *autogen.sh* script creates a set of automatic configuration scripts that will interrogate your system to discover the correct settings and ensure you have all the necessary libraries to compile the code. The most important of these is the configure script that offers a number of different options to customize the build process. Type **./configure --help** to see the various options:

```
$ ./configure --help
`configure' configures Bitcoin Core 0.15.0 to adapt to many kinds of systems.

Usage: ./configure [OPTION]... [VAR=VALUE]...

...
Optional Features:
  --disable-option-checking  ignore unrecognized --enable/--with options
  --disable-FEATURE       do not include FEATURE (same as --enable-FEATURE=no)
  --enable-FEATURE[=ARG]  include FEATURE [ARG=yes]

  --enable-wallet         enable wallet (default is yes)

  --with-gui[=no|qt4|qt5|auto]
...
```

The configure script allows you to enable or disable certain features of bitcoind through the use of the --enable-FEATURE and --disable-FEATURE flags, where `FEATURE` is replaced by the feature name, as listed in the help output. In this chapter, we will build the bitcoind client with all the default features. We won't be using the configuration flags, but you should review them to understand what optional features are part of the client. If you are in an academic setting, computer lab restrictions may require you to install applications in your home directory (e.g., using --prefix=$HOME).

Here are some useful options that override the default behavior of the configure script:

```
<dl>
<dt><code>--prefix=$HOME</code></dt>
<dd><p>This overrides the default installation location (which is
<em>/usr/local/</em>) for the resulting executable. Use <code>$HOME</code> to put
everything in your home directory, or a different path.</p></dd>

<dt><code>--disable-wallet</code></dt>
<dd><p>This is used to disable the reference wallet implementation.</p></dd>

<dt><code>--with-incompatible-bdb</code></dt>
<dd><p>If you are building a wallet, allow the use of an incompatible version of the
Berkeley DB library.</p></dd>

<dt><code>--with-gui=no</code></dt>
<dd><p>Don't build the graphical user interface, which requires the Qt library. This
builds server and command-line bitcoin only.</p></dd>
</dl>
```

Next, run the configure script to automatically discover all the necessary libraries and create a customized build script for your system:

```
$ ./configure
checking build system type... x86_64-unknown-linux-gnu
checking host system type... x86_64-unknown-linux-gnu
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
...
[many pages of configuration tests follow]
...
$
```

If all went well, the configure command will end by creating the customized build scripts that will allow us to compile bitcoind. If there are any missing libraries or errors, the configure command will terminate with an error instead of creating the build scripts. If an error occurs, it is most likely because of a missing or incompatible library. Review the build documentation again and make sure you install the missing prerequisites. Then run configure again and see if that fixes the error.

Building the Bitcoin Core Executables

Next, you will compile the source code, a process that can take up to an hour to complete, depending on the speed of your CPU and available memory. During the compilation process you should see output every few seconds or every few minutes, or an error if something goes wrong. If an error occurs, or the compilation process is interrupted, it can be resumed any time by typing make again. Type **make** to start compiling the executable application:

```
$ make
Making all in src
  CXX      crypto/libbitcoinconsensus_la-hmac_sha512.lo
  CXX      crypto/libbitcoinconsensus_la-ripemd160.lo
  CXX      crypto/libbitcoinconsensus_la-sha1.lo
  CXX      crypto/libbitcoinconsensus_la-sha256.lo
  CXX      crypto/libbitcoinconsensus_la-sha512.lo
  CXX      libbitcoinconsensus_la-hash.lo
  CXX      primitives/libbitcoinconsensus_la-transaction.lo
  CXX      libbitcoinconsensus_la-pubkey.lo
  CXX      script/libbitcoinconsensus_la-bitcoinconsensus.lo
  CXX      script/libbitcoinconsensus_la-interpreter.lo

[... many more compilation messages follow ...]

$
```

On a fast system with more than one CPU, you might want to set the number of parallel compile jobs. For instance, make -j 2 will use two cores if they are available. If all goes well, Bitcoin Core is now compiled. You should run the unit test suite with make check to ensure the linked libraries are not broken in obvious ways. The final step is to install the various executables on your system using the make install command. You may be prompted for your user password, because this step requires administrative privileges:

```
$ make check && sudo make install
Password:
Making install in src
 ../build-aux/install-sh -c -d '/usr/local/lib'
libtool: install: /usr/bin/install -c bitcoind /usr/local/bin/bitcoind
libtool: install: /usr/bin/install -c bitcoin-cli /usr/local/bin/bitcoin-cli
libtool: install: /usr/bin/install -c bitcoin-tx /usr/local/bin/bitcoin-tx
...
$
```

The default installation of bitcoind puts it in */usr/local/bin*. You can confirm that Bitcoin Core is correctly installed by asking the system for the path of the executables, as follows:

```
$ which bitcoind
/usr/local/bin/bitcoind

$ which bitcoin-cli
/usr/local/bin/bitcoin-cli
```

Running a Bitcoin Core Node

Bitcoin’s peer-to-peer network is composed of network "nodes," run mostly by volunteers and some

of the businesses that build bitcoin applications. Those running bitcoin nodes have a direct and authoritative view of the bitcoin blockchain, with a local copy of all the transactions, independently validated by their own system. By running a node, you don't have to rely on any third party to validate a transaction. Moreover, by running a bitcoin node you contribute to the bitcoin network by making it more robust.

Running a node, however, requires a permanently connected system with enough resources to process all bitcoin transactions. Depending on whether you choose to index all transactions and keep a full copy of the blockchain, you may also need a lot of disk space and RAM. As of early 2021, a full-index node needs 2 GB of RAM and a minimum of 360 GB of disk space (see https://blockchain.info/charts/blocks-size). Bitcoin nodes also transmit and receive bitcoin transactions and blocks, consuming internet bandwidth. If your internet connection is limited, has a low data cap, or is metered (charged by the gigabit), you should probably not run a bitcoin node on it, or run it in a way that constrains its bandwidth (see Sample configuration of a resource-constrained system).

TIP

Bitcoin Core keeps a full copy of the blockchain by default, with every transaction that has ever occurred on the bitcoin network since its inception in 2009. This dataset is dozens of gigabytes in size and is downloaded incrementally over several days or weeks, depending on the speed of your CPU and internet connection. Bitcoin Core will not be able to process transactions or update account balances until the full blockchain dataset is downloaded. Make sure you have enough disk space, bandwidth, and time to complete the initial synchronization. You can configure Bitcoin Core to reduce the size of the blockchain by discarding old blocks (see Sample configuration of a resource-constrained system), but it will still download the entire dataset before discarding data.

Despite these resource requirements, thousands of volunteers run bitcoin nodes. Some are running on systems as simple as a Raspberry Pi (a $35 USD computer the size of a pack of cards). Many volunteers also run bitcoin nodes on rented servers, usually some variant of Linux. A *Virtual Private Server* (VPS) or *Cloud Computing Server* instance can be used to run a bitcoin node. Such servers can be rented for $25 to $50 USD per month from a variety of providers.

Why would you want to run a node? Here are some of the most common reasons:

- If you are developing bitcoin software and need to rely on a bitcoin node for programmable (API) access to the network and blockchain.
- If you are building applications that must validate transactions according to bitcoin's consensus rules. Typically, bitcoin software companies run several nodes.
- If you want to support bitcoin. Running a node makes the network more robust and able to serve more wallets, more users, and more transactions.
- If you do not want to rely on any third party to process or validate your transactions.

If you're reading this book and interested in developing bitcoin software, you should be running your own node.

Configuring the Bitcoin Core Node

Bitcoin Core will look for a configuration file in its data directory on every start. In this section we will examine the various configuration options and set up a configuration file. To locate the configuration file, run bitcoind -printtoconsole in your terminal and look for the first couple of lines.

```
$ bitcoind -printtoconsole
Bitcoin version v0.15.0
Using the 'standard' SHA256 implementation
Using data directory /home/ubuntu/.bitcoin/
Using config file /home/ubuntu/.bitcoin/bitcoin.conf
...
[a lot more debug output]
...
```

You can hit Ctrl-C to shut down the node once you determine the location of the config file. Usually the configuration file is inside the *.bitcoin* data directory under your user's home directory. It is not created automatically, but you can create a starter config file by copying and pasting from the Sample configuration of a full-index node example, below. You can create or modify the configuration file in your preferred editor.

Bitcoin Core offers more than 100 configuration options that modify the behavior of the network node, the storage of the blockchain, and many other aspects of its operation. To see a listing of these options, run bitcoind --help:

```
$ bitcoind --help
Bitcoin Core Daemon version v0.15.0

Usage:
  bitcoind [options]                     Start Bitcoin Core Daemon

Options:

  -?
       Print this help message and exit

  -version
       Print version and exit

  -alertnotify=<cmd>
       Execute command when a relevant alert is received or we see a really
       long fork (%s in cmd is replaced by message)
...
[many more options]
...

  -rpcthreads=<n>
       Set the number of threads to service RPC calls (default: 4)
```

Here are some of the most important options that you can set in the configuration file, or as command-line parameters to bitcoind:

alertnotify

Run a specified command or script to send emergency alerts to the owner of this node, usually by email.

conf

An alternative location for the configuration file. This only makes sense as a command-line parameter to bitcoind, as it can't be inside the configuration file it refers to.

datadir

Select the directory and filesystem in which to put all the blockchain data. By default this is the *.bitcoin* subdirectory of your home directory. Make sure this filesystem has several gigabytes of free space.

prune

Reduce the disk space requirements to this many megabytes, by deleting old blocks. Use this on a resource-constrained node that can't fit the full blockchain.

txindex

Maintain an index of all transactions. This means a complete copy of the blockchain that allows you to programmatically retrieve any transaction by ID.

dbcache

The size of the UTXO cache. The default is 450 MiB. Increase this on high-end hardware and reduce the size on low-end hardware to save memory at the expense of slow disk IO.

maxconnections

Set the maximum number of nodes from which to accept connections. Reducing this from the default will reduce your bandwidth consumption. Use if you have a data cap or pay by the gigabyte.

maxmempool

Limit the transaction memory pool to this many megabytes. Use it to reduce memory use on memory-constrained nodes.

maxreceivebuffer/maxsendbuffer

Limit per-connection memory buffer to this many multiples of 1000 bytes. Use on memory-constrained nodes.

minrelaytxfee

Set the minimum fee rate for transaction you will relay. Below this value, the transaction is treated nonstandard, rejected from the transaction pool and not relayed.

Transaction Database Index and txindex Option

By default, Bitcoin Core builds a database containing *only* the transactions related to the user's wallet. If you want to be able to access *any* transaction with commands like getrawtransaction (see Exploring and Decoding Transactions), you need to configure Bitcoin Core to build a complete transaction index, which can be achieved with the txindex option. Set txindex=1 in the Bitcoin Core configuration file. If you don't set this option at first and later set it to full indexing, you need to restart bitcoind with the -reindex option and wait for it to rebuild the index.

Sample configuration of a full-index node shows how you might combine the preceding options, with a fully indexed node, running as an API backend for a bitcoin application.

Example 1. Sample configuration of a full-index node

```
alertnotify=myemailscript.sh "Alert: %s"
datadir=/lotsofspace/bitcoin
txindex=1
```

Sample configuration of a resource-constrained system shows a resource-constrained node running on a smaller server.

Example 2. Sample configuration of a resource-constrained system

```
alertnotify=myemailscript.sh "Alert: %s"
maxconnections=15
prune=5000
dbcache=150
maxmempool=150
maxreceivebuffer=2500
maxsendbuffer=500
```

Once you've edited the configuration file and set the options that best represent your needs, you can test bitcoind with this configuration. Run Bitcoin Core with the option printtoconsole to run in the foreground with output to the console:

```
$ bitcoind -printtoconsole

Bitcoin version v0.15.0
InitParameterInteraction: parameter interaction: -whitelistforcerelay=1 -> setting
-whitelistrelay=1
Assuming ancestors of block
000000000000000003b9ce759c2a087d52abc4266f8f4ebd6d768b89defa50a have valid
signatures.
Using the 'standard' SHA256 implementation
Default data directory /home/ubuntu/.bitcoin
Using data directory /lotsofspace/.bitcoin
Using config file /home/ubuntu/.bitcoin/bitcoin.conf
Using at most 125 automatic connections (1048576 file descriptors available)
Using 16 MiB out of 32/2 requested for signature cache, able to store 524288 elements
Using 16 MiB out of 32/2 requested for script execution cache, able to store 524288
elements
Using 2 threads for script verification
HTTP: creating work queue of depth 16
No rpcpassword set - using random cookie authentication
Generated RPC authentication cookie /lotsofspace/.bitcoin/.cookie
HTTP: starting 4 worker threads
init message: Verifying wallet(s)...
Using BerkeleyDB version Berkeley DB 4.8.30: (April  9, 2010)
Using wallet wallet.dat
CDBEnv::Open: LogDir=/lotsofspace/.bitcoin/database
ErrorFile=/lotsofspace/.bitcoin/db.log
scheduler thread start
Cache configuration:
* Using 250.0MiB for block index database
* Using 8.0MiB for chain state database
* Using 1742.0MiB for in-memory UTXO set (plus up to 286.1MiB of unused mempool space)
init message: Loading block index...
Opening LevelDB in /lotsofspace/.bitcoin/blocks/index
Opened LevelDB successfully

[... more startup messages ...]
```

You can hit Ctrl-C to interrupt the process once you are satisfied that it is loading the correct settings and running as you expect.

To run Bitcoin Core in the background as a process, start it with the daemon option, as bitcoind -daemon.

To monitor the progress and runtime status of your bitcoin node, use the command bitcoin-cli getblockchaininfo:

```
$ bitcoin-cli getblockchaininfo
```

```
{
  "chain": "main",
  "blocks": 0,
  "headers": 83999,
  "bestblockhash": "000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f",
  "difficulty": 1,
  "mediantime": 1231006505,
  "verificationprogress": 3.783041623201835e-09,
  "chainwork": "0000000000000000000000000000000000000000000000000100010001",
  "pruned": false,
  [...]
}
```

This shows a node with a blockchain height of 0 blocks and 83999 headers. The node currently fetches the block headers of the best chain and afterward continues to download the full blocks.

Once you are happy with the configuration options you have selected, you should add bitcoin to the startup scripts in your operating system, so that it runs continuously and restarts when the operating system restarts. You will find a number of example startup scripts for various operating systems in bitcoin's source directory under *contrib/init* and a *README.md* file showing which system uses which script.

Bitcoin Core Application Programming Interface (API)

The Bitcoin Core client implements a JSON-RPC interface that can also be accessed using the command-line helper bitcoin-cli. The command line allows us to experiment interactively with the capabilities that are also available programmatically via the API. To start, invoke the help command to see a list of the available bitcoin RPC commands:

```
$ bitcoin-cli help
addmultisigaddress nrequired ["key",...] ( "account" )
addnode "node" "add|remove|onetry"
backupwallet "destination"
createmultisig nrequired ["key",...]
createrawtransaction [{"txid":"id","vout":n},...] {"address":amount,...}
decoderawtransaction "hexstring"
...
...
verifymessage "bitcoinaddress" "signature" "message"
walletlock
walletpassphrase "passphrase" timeout
walletpassphrasechange "oldpassphrase" "newpassphrase"
```

Each of these commands may take a number of parameters. To get additional help, a detailed description, and information on the parameters, add the command name after help. For example, to see help on the getblockhash RPC command:

```
$ bitcoin-cli help getblockhash
getblockhash height

Returns hash of block in best-block-chain at height provided.

Arguments:
1. height         (numeric, required) The height index

Result:
"hash"         (string) The block hash

Examples:
> bitcoin-cli getblockhash 1000
> curl --user myusername --data-binary '{"jsonrpc": "1.0", "id":"curltest", "method":
"getblockhash", "params": [1000] }' -H 'content-type: text/plain;'
http://127.0.0.1:8332/
```

At the end of the help information you will see two examples of the RPC command, using the bitcoin-cli helper or the HTTP client curl. These examples demonstrate how you might call the command. Copy the first example and see the result:

```
$ bitcoin-cli getblockhash 1000
00000000c937983704a73af28acdec37b049d214adbda81d7e2a3dd146f6ed09
```

The result is a block hash, which is described in more detail in the following chapters. But for now, this command should return the same result on your system, demonstrating that your Bitcoin Core node is running, is accepting commands, and has information about block 1000 to return to you.

In the next sections we will demonstrate some very useful RPC commands and their expected output.

Getting Information on the Bitcoin Core Client Status

Bitcoin Core provides status reports on different modules through the JSON-RPC interface. The most important commands include getblockchaininfo, getmempoolinfo, getnetworkinfo and getwalletinfo.

Bitcoin's getblockchaininfo RPC command was introduced earlier. The getnetworkinfo command displays basic information about the status of the bitcoin network node. Use bitcoin-cli to run it:

```
$ bitcoin-cli getnetworkinfo
```

```
  "version": 150000,
  "subversion": "/Satoshi:0.15.0/",
  "protocolversion": 70015,
  "localservices": "000000000000000d",
  "localrelay": true,
  "timeoffset": 0,
  "networkactive": true,
  "connections": 8,
  "networks": [
    ...
    detailed information about all networks (ipv4, ipv6 or onion)
    ...
  ],
  "relayfee": 0.00001000,
  "incrementalfee": 0.00001000,
  "localaddresses": [
  ],
  "warnings": ""
}
```

The data is returned in JavaScript Object Notation (JSON), a format that can easily be "consumed" by all programming languages but is also quite human-readable. Among this data we see the version numbers for the bitcoin software client (150000) and bitcoin protocol (70015). We see the current number of connections (8) and various information about the bitcoin network and the settings related to this client.

TIP	It will take some time, perhaps more than a day, for the bitcoind client to "catch up" to the current blockchain height as it downloads blocks from other bitcoin clients. You can check its progress using getblockchaininfo to see the number of known blocks.

Exploring and Decoding Transactions

Commands: getrawtransaction, decoderawtransaction

In [cup_of_coffee], Alice bought a cup of coffee from Bob's Cafe. Her transaction was recorded on the blockchain with transaction ID (txid) 0627052b6f28912f2703066a912ea577f2ce4da4caa5a5fbd8a57286c345c2f2. Let's use the API to retrieve and examine that transaction by passing the transaction ID as a parameter:

```
<pre data-type="programlisting">
$ bitcoin-cli getrawtransaction 0627052b6f28912f2703066a912ea577f2ce4da4caa5a&#x21b5;
5fbd8a57286c345c2f2

0100000001186f9f998a5aa6f048e51dd8419a14d8a0f1a8a2836dd734d2804fe65fa35779000&#x21b5;
000008b483045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4&#x21b5;
ae24cb02204b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813014&#x21b5;
10484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe423cc54123363767&#x21b5;
89d172787ec3457eee41c04f4938de5cc17b4a10fa336a8d752adfffffffff0260e3160000000&#x21b5;
0001976a914ab68025513c3dbd2f7b92a94e0581f5d50f654e788acd0ef8000000000001976a9&#x21b5;
147f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a888ac00000000
</pre>
```

TIP A transaction ID is not authoritative until a transaction has been confirmed. Absence of a transaction hash in the blockchain does not mean the transaction was not processed. This is known as "transaction malleability," because transaction hashes can be modified prior to confirmation in a block. After confirmation, the txid is immutable and authoritative.

The command getrawtransaction returns a serialized transaction in hexadecimal notation. To decode that, we use the decoderawtransaction command, passing the hex data as a parameter. You can copy the hex returned by getrawtransaction and paste it as a parameter to decoderawtransaction:

```
<pre data-type="programlisting">
$ bitcoin-cli decoderawtransaction 0100000001186f9f998a5aa6f048e51dd8419a14d8&#x21b5;
a0f1a8a2836dd734d2804fe65fa35779000000008b483045022100884d142d86652a3f47ba474&#x21b5;
6ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039ff08df09cbe9f6addac960298&#x21b5;
cad530a863ea8f53982c09db8f6e381301410484ecc0d46f1918b30928fa0e4ed99f16a0fb4fd&#x21b5;
e0735e7ade8416ab9fe423cc5412336376789d172787ec3457eee41c04f4938de5cc17b4a10fa&#x21b5;
336a8d752adfffffffff0260e31600000000001976a914ab68025513c3dbd2f7b92a94e0581f5&#x21b5;
d50f654e788acd0ef8000000000001976a9147f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8&#x21b5;
88ac00000000

</pre>
```

```
<pre data-type="programlisting" data-code-language="json">
{
  "txid": "0627052b6f28912f2703066a912ea577f2ce4da4caa5a5fbd8a57286c345c2f2",
  "size": 258,
  "version": 1,
  "locktime": 0,
  "vin": [
    {
      "txid": "7957a35fe64f80d234d76d83a2...8149a41d81de548f0a65a8a999f6f18",
      "vout": 0,
      "scriptSig": {
        "asm":"3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1decc...",
        "hex":"483045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1de..."
      },
      "sequence": 4294967295
    }
  ],
  "vout": [
    {
      "value": 0.01500000,
      "n": 0,
      "scriptPubKey": {
        "asm": "OP_DUP OP_HASH160 ab68...5f654e7 OP_EQUALVERIFY OP_CHECKSIG",
        "hex": "76a914ab68025513c3dbd2f7b92a94e0581f5d50f654e788ac",
        "reqSigs": 1,
        "type": "pubkeyhash",
        "addresses": [
          "1GdK9UzpHBzqzX2A9JFP3Di4weBwqgmoQA"
        ]
      }
    },
    {
      "value": 0.08450000,
      "n": 1,
      "scriptPubKey": {
        "asm": "OP_DUP OP_HASH160 7f9b1a...025a8 OP_EQUALVERIFY OP_CHECKSIG",
        "hex": "76a9147f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a888ac",
        "reqSigs": 1,
        "type": "pubkeyhash",
        "addresses": [
          "1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK"
        ]
      }
    }
  ]
}

</pre>
```

The transaction decode shows all the components of this transaction, including the transaction

inputs and outputs. In this case we see that the transaction that credited our new address with 15 millibits used one input and generated two outputs. The input to this transaction was the output from a previously confirmed transaction (shown as the vin txid starting with 7957a35fe). The two outputs correspond to the 15 millibit credit and an output with change back to the sender.

We can further explore the blockchain by examining the previous transaction referenced by its txid in this transaction using the same commands (e.g., getrawtransaction). Jumping from transaction to transaction we can follow a chain of transactions back as the coins are transmitted from owner address to owner address.

Exploring Blocks

Commands: getblock, getblockhash

Exploring blocks is similar to exploring transactions. However, blocks can be referenced either by the block *height* or by the block *hash*. First, let's find a block by its height. In [cup_of_coffee], we saw that Alice's transaction was included in block 277316.

We use the getblockhash command, which takes the block height as the parameter and returns the block hash for that block:

```
<pre data-type="programlisting">
$ bitcoin-cli getblockhash 277316
000000000000001b6b9a13b095e96db41c4a928b97ef2d944a9b31b2cc7bdc4
</pre>
```

Now that we know which block Alice's transaction was included in, we can query that block. We use the getblock command with the block hash as the parameter:

```
<pre data-type="programlisting">
$ bitcoin-cli getblock 000000000000001b6b9a13b095e96db41c4a928b97ef2d944a9b3&#x21b5;
1b2cc7bdc4
</pre>
```

```
<pre data-type="programlisting" data-code-language="json">
{
  "hash": "000000000000001b6b9a13b095e96db41c4a928b97ef2d944a9b31b2cc7bdc4",
  "confirmations": 37371,
  "size": 218629,
  "height": 277316,
  "version": 2,
  "merkleroot": "c91c008c26e50763e9f548bb8b2fc323735f73577effbc55502c51eb4cc7cf2e",
  "tx": [
    "d5ada064c6417ca25c4308bd158c34b77e1c0eca2a73cda16c737e7424afba2f",
    "b268b45c59b39d759614757718b9918caf0ba9d97c56f3b91956ff877c503fbe",
    "04905ff987ddd4cfe603b03cfb7ca50ee81d89d1f8f5f265c38f763eea4a21fd",
    "32467aab5d04f51940075055c2f20bbd1195727c961431bf0aff8443f9710f81",
    "561c5216944e21fa29dd12aaa1a45e3397f9c0d888359cb05e1f79fe73da37bd",
[... hundreds of transactions ...]
    "78b300b2a1d2d9449b58db7bc71c3884d6e0579617e0da4991b9734cef7ab23a",
    "6c87130ec283ab4c2c493b190c20de4b28ff3caf72d16ffa1ce3e96f2069aca9",
    "6f423dbc3636ef193fd8898dfdf7621dcade1bbe509e963ffbff91f696d81a62",
    "802ba8b2adabc5796a9471f25b02ae6aeee2439c679a5c33c4bbcee97e081196",
    "eaaf6a048588d9ad4d1c092539bd571dd8af30635c152a3b0e8b611e67d1a1af",
    "e67abc6bd5e2cac169821afc51b207127f42b92a841e976f9b752157879ba8bd",
    "d38985a6a1bfd35037cb7776b2dc86797abbb7a06630f5d03df2785d50d5a2ac",
    "45ea0a3f6016d2bb90ab92c34a7aac9767671a8a84b9bcce6c019e60197c134b",
    "c098445d748ced5f178ef2ff96f2758cbec9eb32cb0fc65db313bcac1d3bc98f"
  ],
  "time": 1388185914,
  "mediantime": 1388183675,
  "nonce": 924591752,
  "bits": "1903a30c",
  "difficulty": 1180923195.258026,
  "chainwork": "0000000000000000000000000000000000000000000934695e92aaf53afa1a",
  "previousblockhash":
"0000000000000002a7bbd25a417c0374cc55261021e8a9ca74442b01284f0569",
  "nextblockhash": "000000000000000010236c269dd6ed714dd5db39d36b33959079d78dfd431ba7"
}
</pre>
```

The block contains 419 transactions and the 64th transaction listed (0627052b...) is Alice's coffee payment. The height entry tells us this is the 277316th block in the blockchain.

Using Bitcoin Core's Programmatic Interface

The bitcoin-cli helper is very useful for exploring the Bitcoin Core API and testing functions. But the whole point of an application programming interface is to access functions programmatically. In this section we will demonstrate accessing Bitcoin Core from another program.

Bitcoin Core's API is a JSON-RPC interface. JSON stands for JavaScript Object Notation and it is a very convenient way to present data that both humans and programs can easily read. RPC stands for Remote Procedure Call, which means that we are calling procedures (functions) that are remote

(on the Bitcoin Core node) via a network protocol. In this case, the network protocol is HTTP, or HTTPS (for encrypted connections).

When we used the bitcoin-cli command to get help on a command, it showed us an example of using curl, the versatile command-line HTTP client to construct one of these JSON-RPC calls:

```
$ curl --user myusername --data-binary '{"jsonrpc": "1.0", "id":"curltest", "method":
"getblockchaininfo", "params": [] }' -H 'content-type: text/plain;'
http://127.0.0.1:8332/
```

This command shows that curl submits an HTTP request to the local host (127.0.0.1), connecting to the default bitcoin port (8332), and submitting a jsonrpc request for the getblockchaininfo method using text/plain encoding.

You might notice that curl will ask for credentials to be sent along with the request. Bitcoin Core will create a random password on each start and place it in the data directory under the name .cookie. The bitcoin-cli helper can read this password file given the data directory. Similarly, you can copy the password and pass it to curl (or any higher level Bitcoin Core RPC wrappers). Alternatively, you can create a static password with the helper script provided in *./share/rpcauth/rpcauth.py* in Bitcoin Core's source directory.

If you're implementing a JSON-RPC call in your own program, you can use a generic HTTP library to construct the call, similar to what is shown in the preceding curl example.

However, there are libraries in most every programming language that "wrap" the Bitcoin Core API in a way that makes this a lot simpler. We will use the python-bitcoinlib library to simplify API access. Remember, this requires you to have a running Bitcoin Core instance, which will be used to make JSON-RPC calls.

The Python script in Running getblockchaininfo via Bitcoin Core's JSON-RPC API makes a simple getblockchaininfo call and prints the blocks parameter from the data returned by Bitcoin Core (full node required).

Example 3. Running getblockchaininfo via Bitcoin Core's JSON-RPC API

```
from bitcoin.rpc import RawProxy

# Create a connection to local Bitcoin Core node
p = RawProxy()

# Run the getblockchaininfo command, store the resulting data in info
info = p.getblockchaininfo()

# Retrieve the 'blocks' element from the info
print(info['blocks'])
```

Running it gives us the following result:

```
$ python rpc_example.py
394075
```

It tells us that our local Bitcoin Core node has 394075 blocks in its blockchain. Not a spectacular result, but it demonstrates the basic use of the library as a simplified interface to Bitcoin Core's JSON-RPC API.

Next, let's use the getrawtransaction and decodetransaction calls to retrieve the details of Alice's coffee payment. In Retrieving a transaction and iterating its outputs, we retrieve Alice's transaction and list the transaction's outputs. For each output, we show the recipient address and value. As a reminder, Alice's transaction had one output paying Bob's Cafe and one output for change back to Alice.

Example 4. Retrieving a transaction and iterating its outputs

```
from bitcoin.rpc import RawProxy

p = RawProxy()

# Alice's transaction ID
txid = "0627052b6f28912f2703066a912ea577f2ce4da4caa5a5fbd8a57286c345c2f2"

# First, retrieve the raw transaction in hex
raw_tx = p.getrawtransaction(txid)

# Decode the transaction hex into a JSON object
decoded_tx = p.decoderawtransaction(raw_tx)

# Retrieve each of the outputs from the transaction
for output in decoded_tx['vout']:
    print(output['scriptPubKey']['addresses'], output['value'])
```

Running this code, we get:

```
$ python rpc_transaction.py
([u'1GdK9UzpHBzqzX2A9JFP3Di4weBwqgmoQA'], Decimal('0.01500000'))
([u'1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK'], Decimal('0.08450000'))
```

Both of the preceding examples are rather simple. You don't really need a program to run them; you could just as easily use the bitcoin-cli helper. The next example, however, requires several hundred RPC calls and more clearly demonstrates the use of a programmatic interface.

In Retrieving a block and adding all the transaction outputs, we first retrieve block 277316, then retrieve each of the 419 transactions within by reference to each transaction ID. Next, we iterate through each of the transaction's outputs and add up the value.

Example 5. Retrieving a block and adding all the transaction outputs

```
from bitcoin.rpc import RawProxy

p = RawProxy()

# The block height where Alice's transaction was recorded
blockheight = 277316

# Get the block hash of block with height 277316
blockhash = p.getblockhash(blockheight)

# Retrieve the block by its hash
block = p.getblock(blockhash)

# Element tx contains the list of all transaction IDs in the block
transactions = block['tx']

block_value = 0

# Iterate through each transaction ID in the block
for txid in transactions:
    tx_value = 0
    # Retrieve the raw transaction by ID
    raw_tx = p.getrawtransaction(txid)
    # Decode the transaction
    decoded_tx = p.decoderawtransaction(raw_tx)
    # Iterate through each output in the transaction
    for output in decoded_tx['vout']:
        # Add up the value of each output
        tx_value = tx_value + output['value']

    # Add the value of this transaction to the total
    block_value = block_value + tx_value

print("Total value in block: ", block_value)
```

Running this code, we get:

```
$ python rpc_block.py

('Total value in block: ', Decimal('10322.07722534'))
```

Our example code calculates that the total value transacted in this block is 10,322.07722534 BTC (including 25 BTC reward and 0.0909 BTC in fees). Compare that to the amount reported by a block explorer site by searching for the block hash or height. Some block explorers report the total value excluding the reward and excluding the fees. See if you can spot the difference.

Alternative Clients, Libraries, and Toolkits

There are many alternative clients, libraries, toolkits, and even full-node implementations in the bitcoin ecosystem. These are implemented in a variety of programming languages, offering programmers native interfaces in their preferred language.

The following sections list some of the best libraries, clients, and toolkits, organized by programming languages.

C/C++

Bitcoin Core

The reference implementation of bitcoin

libbitcoin

Cross-platform C++ development toolkit, node, and consensus library

bitcoin explorer

Libbitcoin's command-line tool

picocoin

A C language lightweight client library for bitcoin by Jeff Garzik

JavaScript

bcoin

A modular and scalable full-node implementation with API

Bitcore

Full node, API, and library by Bitpay

BitcoinJS

A pure JavaScript Bitcoin library for node.js and browsers

Java

bitcoinj

A Java full-node client library

PHP

bitwasp/bitcoin

A PHP bitcoin library, and related projects

Python

python-bitcoinlib

A Python bitcoin library, consensus library, and node by Peter Todd

pycoin

A Python bitcoin library by Richard Kiss

pybitcointools

An archived fork of Python bitcoin library by Vitalik Buterin

Ruby

bitcoin-client

A Ruby library wrapper for the JSON-RPC API

Go

btcd

A Go language full-node bitcoin client

Rust

rust-bitcoin

Rust bitcoin library for serialization, parsing, and API calls

C#

NBitcoin

Comprehensive bitcoin library for the .NET framework

Objective-C

CoreBitcoin

Bitcoin toolkit for ObjC and Swift

Many more libraries exist in a variety of other programming languages and more are created all the time.