ElectrumSV node project

The ElectrumSV developers

Jun 01, 2021

USING OUR BUILDS

The ElectrumSV node project is intended to provide cross-platform and easily runnable builds of the node software for developers to develop, test and experiment with. While ElectrumSV wallet uses these builds for it's automated testing processes, and in other situations, these builds are not ElectrumSV-specific in any way. However, as ElectrumSV is a trusted project in the Bitcoin SV ecosystem, by curating these builds we can provide them in a form developers can trust.

You are more than welcome to get involved and help us make our builds even better.

The goal of the developers of the official Bitcoin SV node project, is to release nodes for miners to use. It would drain their resources to also have to focus on creating cross-platform development environments for application developers. For this reason, they only provide Linux builds themselves.

Important: Head on over to electrumsv.io/node-project to find the node project home page, and links to the latest archived builds.

ONE

POSSIBLE USES

- Run your application against a local blockchain you control, generating both blocks and coins when you need them.

- Stress test your application with continual reorgs, ensuring your handling is robust.

- Develop offline, whether due to travel or productivity reasons, taking advantage of your ability to access a blockchain without network connectivity.

1.1 Python packages

The original reason that these packages are created containing standalone installs of the node software, is that ElectrumSV is written in Python and uses the Python packaging system to acquire it's dependencies. The Python package repository enforces strict compatibility rules on uploaded binary packages like our node packages, and this ensures that those packages will run on as wide a range of computers that use the architectures we support.

As of *electrumsv-node* version 0.0.23 we support the following platforms:

- 64 bit Linux.

- 64 bit MacOS (OS X 10.9 and above).

- 64 bit Windows.

For the following versions of Python:

- Python 3.7

- Python 3.8

- Python 3.9

1.1.1 Installing the package

Different platforms have different ways of installing Python. If you are using Windows or MacOS, you can go to the official Python web site and download a version. If you are using Linux you will need to work out how to install Python for whatever Linux distribution, we cannot help you with that.

Windows

Ensure you have Python 3.7 or later, you can do so with the following command at a DOS prompt:

```
> py --version
3.9.5
```

Ensure your *pip* package installation command is associated with the given version of Python, which in this case is 3.9:

```
> pip --version
pip 21.1.1 from c:\...\python\python39\lib\site-packages\pip (python 3.9)
```

If your *py* command does not find a version of Python that is supported by *electrumsv-node*, try the commands *py –help* and *py –list* to see what your options are.

Now that you know what Python version you are installing the package for, you can go ahead and install it (remember to add any additional arguments you need to *py*):

```
> pip install -U electrumsv-node
```

This should complete successfully, given that you are using 64-bit Windows and a compatible version of Python.

MacOS or Linux

Ensure you have Python 3.7 or later, you can do so with the following command in a terminal:

```
$ python3 --version
Python 3.9.1
```

Ensure that your `pip3` command is associated with the version of Python that you are using. Check the following command prints a message that ends with something like (`Python 3.8`) that matches your Python version:

```
pip3 --version
```

Now you should be able to install the *electrumsv-node* package:

```
pip3 install -U electrumsv-node
```

This should complete successfully, given that you are using a 64-bit processor and a compatible version of Python.

1.1.2 Running the node

Each Python package contains a Python module and the node binaries for the given platform. The module is just a simple wrapper that starts and stops running the node, and allows the user to differentiate the parameters so that they can if necessary run multiple nodes at the same time. As Python packages are just zip archives, it is possible for a user to download a package file and extract the included node binaries if they wish.

Running one node instance

This package is primarily intended to be used to run local regtest nodes. This example will show how to run one regtest node, and how to generate blocks and obtain regtest coins using it. The shown commands will be for Windows, but will work for all supported operating systems.

Starting the node:

```
>>> import electrumsv_node
>>> electrumsv_node.start()
26400
```

The value returned is the process id, which in this case is `26400`. While it is possible to specify that the node runs against testnet or other blockchains, by default as shown here it runs against the regtest blockchain. We will now call an RPC method on the running node, check that we get a response and that our local regtest blockchain is a blank slate with no mined blocks.

Calling the `getinfo` RPC method:

```
>>> result = electrumsv_node.call_any("getinfo")
>>> data = result.json()
>>> data
{'result': {'version': 101000600, 'protocolversion': 70015, 'walletversion': 160300,
→'balance': 0.0, 'blocks': 0, 'timeoffset': 0, 'connections': 0, 'proxy': '',
→'difficulty': 4.656542373906925e-10, 'testnet': False, 'stn': False, 'keypoololdest':
→1622424022, 'keypoolsize': 2000, 'paytxfee': 0.0, 'relayfee': 2.5e-06, 'errors': 'This
→is a pre-release or beta test build - use at your own risk - do not use for mining or
→merchant applications', 'maxblocksize': 10000000000, 'maxminedblocksize': 128000000,
→'maxstackmemoryusagepolicy': 100000000, 'maxstackmemoryusageconsensus':
→9223372036854775807}, 'error': None, 'id': 0}
>>> data["result"]["blocks"]
>>> 0
```

The response indicates that the node is running, that the wallet is compiled into it, and that the no blocks have been mined yet. We will now mine one block to an arbitrary regtest address.

Mining a block:

```
>>> result = electrumsv_node.call_any("generatetoaddress", 1,
→"mfs8Y8gAwC2vJHCeSXkHs6LF5nu5PA7nxc")
>>> result.json()
{'result': ['1e6b5730e742e7d26338384056563bee8a9b0b06f70b279efe899117e2003366'], 'error
→': None, 'id': 0}
```

The response indicates that one block with the given hash was mined. At this point we know how to start a node and call RPC methods, including generating a block to send coins to one of our addresses. It is not intended that this documentation illustrate how to use the node RPC methods, but just to show they can be called in useful ways.

One of the biggest advantages of running a regtest node is that it is your own test blockchain, and you can reset it any time you want, starting again from a fresh blockchain. First let's confirm that the local regtest blockchain has a mined block.

```
>>> result = electrumsv_node.call_any("getinfo")
>>> result.json()["result"]["blocks"]
1
```

Let's reset the blockchain, so that it is a blank slate we can use to do fresh tests against.

Resetting the node:

```
>>> electrumsv_node.reset()
```

With the node reset, we can start it again and confirm that it has been properly reset.

```
>>> electrumsv_node.start()
8208
>>> electrumsv_node.call_any("getinfo").json()["result"]["blocks"]
0
```

Running multiple node instances

There's a few things that happen when you run one node instance that make it a lot easier to do without worrying about the details. This includes using the default values to:

- Specify a unique directory for the node to put blockchain data in.

- Specify which port the node should use for RPC.

- Specify which port the node should use for ZMQ.

- Specify which port the node should use for P2P.

If you are going to run multiple node instances you need to pass unique values for each of these as parameters when calling methods on the `electrum_node` Python module.

Start the two nodes:

```
>>> import os, tempfile
>>> base_temp_path = tempfile.gettempdir()
>>> temp_path1 = os.path.join(base_temp_path, "node1")
>>> P2P_PORT1=8001
>>> ZMQ_PORT1=8011
>>> RPC_PORT1=8021
>>> temp_path2 = os.path.join(base_temp_path, "node2")
>>> P2P_PORT2=8002
>>> ZMQ_PORT2=8012
>>> RPC_PORT2=8022
>>> electrumsv_node.start(rpcport=RPC_PORT1, p2p_port=P2P_PORT1, zmq_port=ZMQ_PORT1,
→data_path=temp_path1)
27792
>>> electrumsv_node.start(rpcport=RPC_PORT2, p2p_port=P2P_PORT2, zmq_port=ZMQ_PORT2,
→data_path=temp_path2)
27608
```

At this stage, both nodes lack any knowledge of any other node. They have no way to know about other nodes they can establish P2P connections to, to share transactions and blocks with.

Tell the first node about the second node:

```
>>> result = electrumsv_node.call_any("addnode", f"127.0.0.1:{P2P_PORT2}", "add",
→rpcport=RPC_PORT1)
>>> result.json()
{'result': None, 'error': None, 'id': 0}
```

The first node will now establish an outgoing P2P connection to the second node. You may need to wait a a little bit for it to happen.

Generate a block on the second node:

```
>>> result = electrumsv_node.call_any("generatetoaddress", 1,
↪"mfs8Y8gAwC2vJHCeSXkHs6LF5nu5PA7nxc", rpcport=RPC_PORT2)
>>> result.json()
{'result': ['7d8c4bd2d396e0f4144560b27579f91045e04fdc61359f94f261276237c9280e'], 'error
↪': None, 'id': 0}
```

Whether your two nodes are connected yet, or are in the process of connecting still, the second node will share this block with the first node shortly after the connection is established.

Check the status of the first node:

```
>>> electrumsv_node.call_any("getinfo", rpcport=RPC_PORT1).json()["result"]["blocks"]
1
```

You can now see that the first node has received the block you had the second node mine. Your local network of nodes is working. A good next step might be to work out how to cause a reorg in one node, by mining a forked longer chain in the other.

1.2 Archived binaries

Originally, the ElectrumSV node project only provided Python packages. However we now also provide the compiled binaries for MacOS and Windows. We do not provide binaries for Linux as you can get them from the official web site, and you should always get your node builds from there if they provide them.

Fig. 1: The Windows archived binaries for a past release.

1.2.1 Verify your download

As always, you should not blindly download executable files from web sites and run them. It is strongly advised you verify these downloaded files before you consider running them.

Important: Bitcoin nodes have been compiled into malware which when deployed onto users computers covertly, mine coins for the malware authors. You may get false positives from your virus checking software for these builds. Check the binaries are signed where applicable, otherwise check the provided checksums for them.

Windows

These binaries are currently signed with the code signing certificate the ElectrumSV developers use for signing the ElectrumSV Windows downloads. You can check these to verify that your downloaded files are authentic and have not been replaced or tampered with. Instructions on how to check our digital signatures are given in the ElectrumSV documentation.
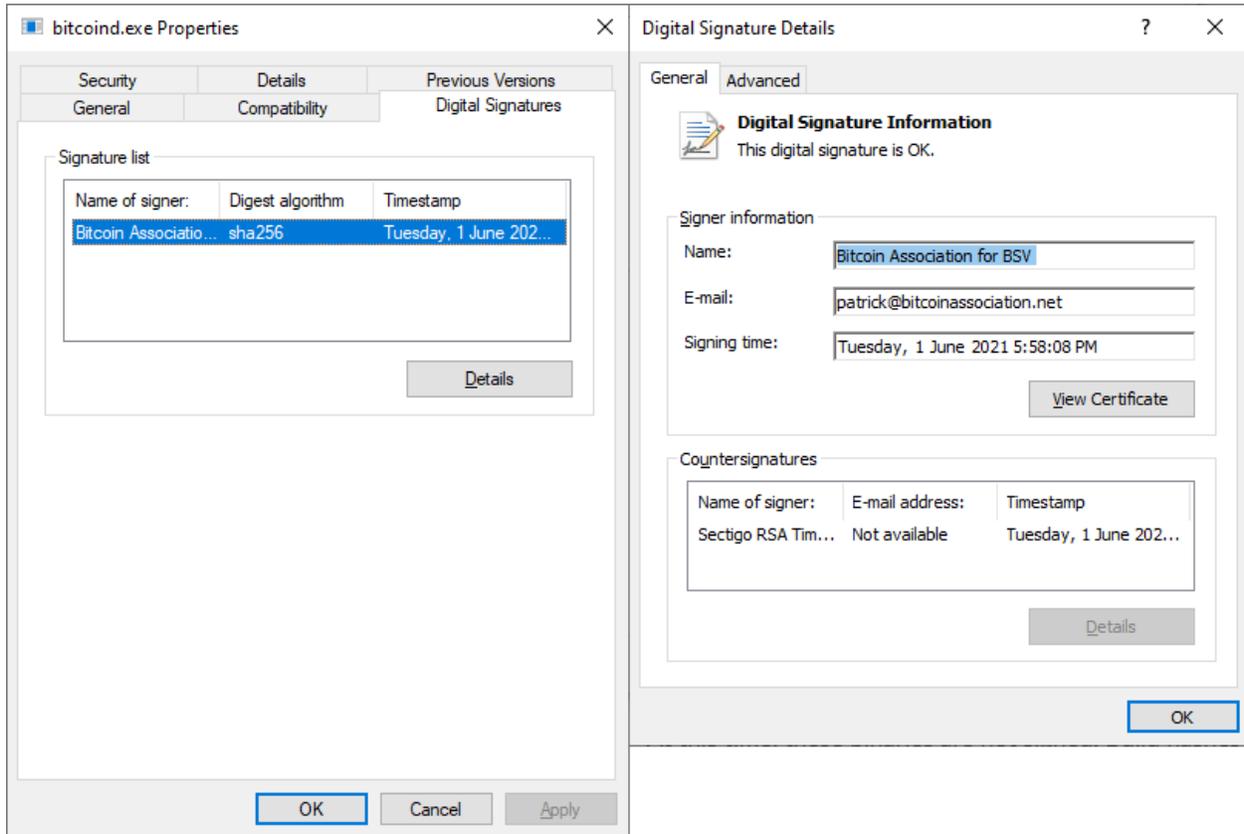

Fig. 2: Checking the digital signature as you would for any signed Windows node binary.

MacOS

At this time, these binaries are not signed. Build hashes will be provided in our Github repository, and can be used to verify that your downloaded files are authentic and have not been replaced or tampered with. Instructions on how to get the checksum for a file you have downloaded are given in the ElectrumSV documentation.

1.2.2 Running the node

Each Python package contains a Python module and the node binaries for the given platform. The module is just a simple wrapper that starts and stops running the node, and allows the user to differentiate the parameters so that they can if necessary run multiple nodes at the same time. As Python packages are just zip archives, it is possible for a user to download a package file and extract the included node binaries if they wish.

Running one node instance

This project is intended to be used to run local regtest nodes. This example will show how to run a regtest node, and how to generate blocks and obtain regtest coins using it. The shown commands will be for Windows, but will work for all supported operating systems.

Create the data directory for your node, let's call it `data_dir1`:

```
> mkdir data_dir1
```

This is where the node will look for the configuration file, and also put it's own data related to the blockchain and the log file you can look to see what is going on.

Create a `data_dir1\bitcoin.conf` file and place the following inside it:

```
server=1
maxstackmemoryusageconsensus=0
excessiveblocksize=10000000000
rpcuser=rpcuser
rpcpassword=rpcpassword
txindex=1
```

The configuration file saves you from having to pass these arguments to `bitcoind` every time you run it.

Starting the node:

```
> bitcoind -regtest -datadir=data_dir1 -rpcport=18322
```

This will do nodey things and prevent any further commands from being entered in the console until you shut down the node.

Calling the `getinfo` RPC method:

```
> bitcoin-cli -datadir=data_dir1 -rpcport=18322 getinfo
{
"version": 101000800,
...
"walletversion": 160300,
...
"blocks": 0,
...
}
```

The response indicates that the node is running, that it is version 1.0.8, that the wallet is compiled into it, and that the no blocks have been mined yet. The *errors* entry is simply because we did not compile the node as a production build, which this project does not intend to provide. We will now mine one block to an arbitrary regtest address.

Mining a block:

```
> bitcoin-cli -datadir=data_dir1 -rpcport=18322 generatetoaddress 1␣
↪mfs8Y8gAwC2vJHCeSXkHs6LF5nu5PA7nxc
[
"42e587c8d433663e2bdacc835aef637527da9a0c36d21f5b1b20b34e2ada86a5"
]
```

The response indicates that one block with the given hash was mined. At this point we know how to start a node and call RPC methods, including generating a block to send coins to one of our addresses. It is not intended that this documentation illustrate how to use the node RPC methods, but just to show they can be called in useful ways.

One of the biggest advantages of running a regtest node is that it is your own test blockchain, and you can reset it any time you want, starting again from a fresh blockchain. First let's confirm that the local regtest blockchain has a mined block.

> bitcoin-cli -datadir=data_dir1 -rpcport=18322 getinfo { … "blocks": 1, … }

The cleanest way to shut down your node, is to tell it to stop with the `bitcoin-cli` command.

Stopping your node cleanly:

```
> bitcoin-cli -datadir=data_dir1 -rpcport=18322 stop
Bitcoin server stopping
```

To reset the blockchain, just delete your data directory. The next time you run the node it will start from scratch with a fresh blockchain.

Running multiple node instances

If you are going to run multiple node instances you need to pass unique values for each of the data directory, the rpc port and the p2p port, for each node.

Edit your `bitcoin.conf` file with the following contents:

```
server=1
maxstackmemoryusageconsensus=0
excessiveblocksize=10000000000
rpcuser=rpcuser
rpcpassword=rpcpassword
txindex=1
```

Make a data directory for each node, `data_dir1` and `data_dir2`:

```
> mkdir data_dir1
> mkdir data_dir2
```

Copy the `bitcoin.conf` file into each data directory:

```
> copy bitcoin.conf data_dir1
> copy bitcoin.conf data_dir2
```

We now need different RPC and P2P ports for each node. The RPC port is the port that we send commands to the node as a user, to direct it what to do. The P2P port is the port that other nodes will connect to a node with. We will use 21011 as the RPC port and 21012 as the P2P port for the first node, and 22011 as the RPC port and 22012 as the P2P port for the second node.

You will need to open two different consoles for the next step, one to start each node in. Remember that the node blocks the console while it does it's nodey things.

Start the first node:

```
> bitcoind -regtest -datadir=data_dir1 -rpcport=21011 -port=21012
```

Start the second node:

```
> bitcoind -regtest -datadir=data_dir2 -rpcport=22011 -port=22012
```

At this stage, both nodes lack any knowledge of any other node. They have no way to know about other nodes they can establish P2P connections to, to share transactions and blocks with. As far as they know they are building their own blockchains in isolation.

We can put them in touch by telling the first node the P2P address (`127.0.0.1:22012`) of the second node.

Tell the first node about the second node:

```
> bitcoin-cli -datadir=data_dir1 -rpcport=21011 addnode 127.0.0.1:22012 add
```

The first node will now establish an outgoing P2P connection to the second node. You may need to wait a a little bit for it to happen. If you want to see what is going on, remember that each node has a log file in their data directory. For the first node this will be `data_dir1\regtest\bitcoind.log`. You should be able to work out where the second node's log file is.

Check if the second node is connected to the first node:

```
> bitcoin-cli -datadir=data_dir2 -rpcport=22011 getinfo
{
...
"connections": 1,
...
}
```

Generate a block on the second node:

```
> bitcoin-cli -datadir=data_dir2 -rpcport=22011 generatetoaddress 1
↪mfs8Y8gAwC2vJHCeSXkHs6LF5nu5PA7nxc
[
"21c15b8b895dac4ccd0ae5790c3a0053fc412881ab999ca16b60df8ccdc9462b"
]
```

Whether your two nodes are connected yet, or are in the process of connecting still, the second node will share this block with the first node shortly after the connection is established.

Check the status of the first node:

```
> bitcoin-cli -datadir=data_dir1 -rpcport=21011 getinfo
{
...
"blocks": 1,
...
}
```

You can now see that the first node has received the block you had the second node mine. Your local network of nodes is working. A good next step might be to work out how to cause a reorg in one node, by mining a forked longer chain in the other.

For now stop your nodes:

```
> bitcoin-cli -datadir=data_dir1 -rpcport=21011 stop
Bitcoin server stopping
> bitcoin-cli -datadir=data_dir2 -rpcport=22011 stop
Bitcoin server stopping
```