CHAPTER 2

Implementing Your Own Blockchain Using Python

In the previous chapter, you learned about the basics of blockchain – the motivations behind blockchains, how transactions are added to blocks, and how blocks are added to the previous blocks to form a chain of blocks called blockchain. A good way to understand all these concepts would be to build one yourself. By implementing your own blockchain, you will have a more detailed look at how concepts like transactions, mining, and consensus work.

Obviously, implementing a full blockchain is not for the faint of heart, and that is definitely not our goal in this chapter. What we will try to do in this chapter is to implement a conceptual blockchain using Python and use it to illustrate the key concepts.

Our Conceptual Blockchain Implementation

For this chapter, we will build a very simple blockchain as shown in Figure 2-1.

W.-M. Lee, *Beginning Ethereum Smart Contracts Programming*,
https://doi.org/10.1007/978-1-4842-5086-0_2

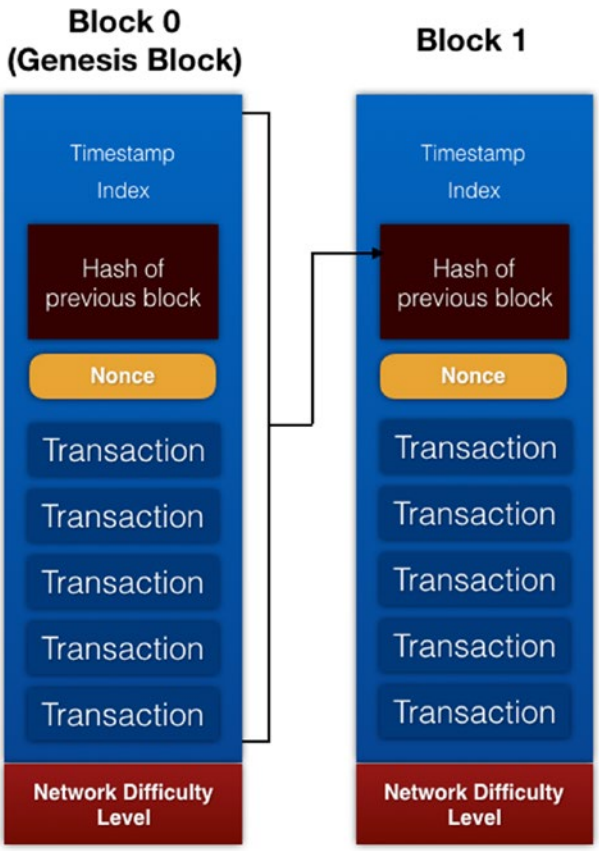

Figure 2-1. *Our conceptual blockchain*

To keep things simple, each block in our blockchain will contain the following components:

- **Timestamp** – the time that the block was added to the blockchain.
- **Index** – a running number starting from 0 indicating the block number.
- **Hash of the previous block** – the hash result of the previous block. As shown in the figure, the hash is the result of hashing the content of the block consisting of the timestamp, index, hash of previous block, nonce, and all the transactions.
- **Nonce** – the *number used once*.
- **Transaction(s)** – each block will hold a variable number of transactions.

Note For simplicity, we are not going to worry about representing the transactions in a Merkle tree, nor are we going to separate a block into block header and content.

The **network difficulty level** will also be fixed at four zeros – that is, in order to derive the nonce, the result of the hash of the block must start with four zeros.

Tip Refer to Chapter 1 for the idea behind nonce and how it relates to network difficulty level.

Obtaining the Nonce

For our sample blockchain implementation, the nonce is found by combining it with the index of the block, hash of the previous block, and all the transactions and checking if the resultant hash matches the network difficulty level (see Figure 2-2).

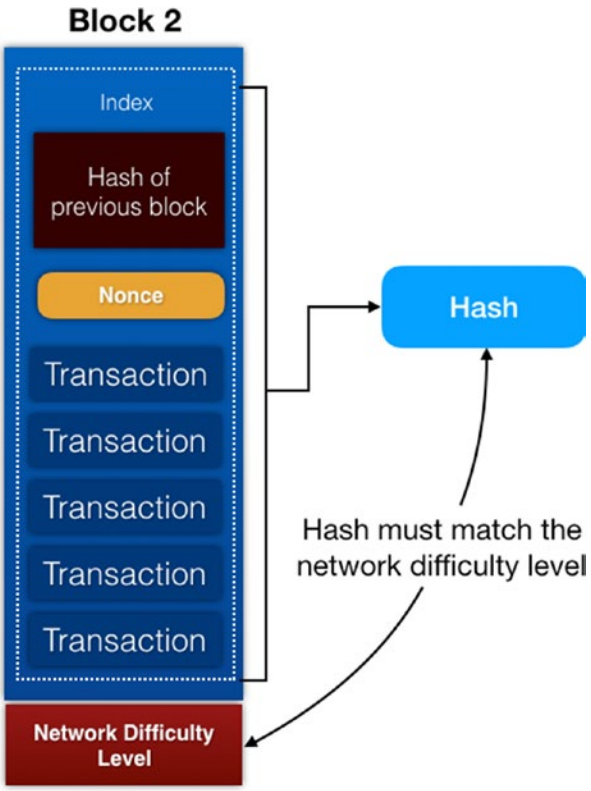

Figure 2-2. *How the nonce will be derived in our conceptual blockchain*

Once the nonce is found, the block will be appended to the last block in the blockchain, with the timestamp added (see Figure 2-3).

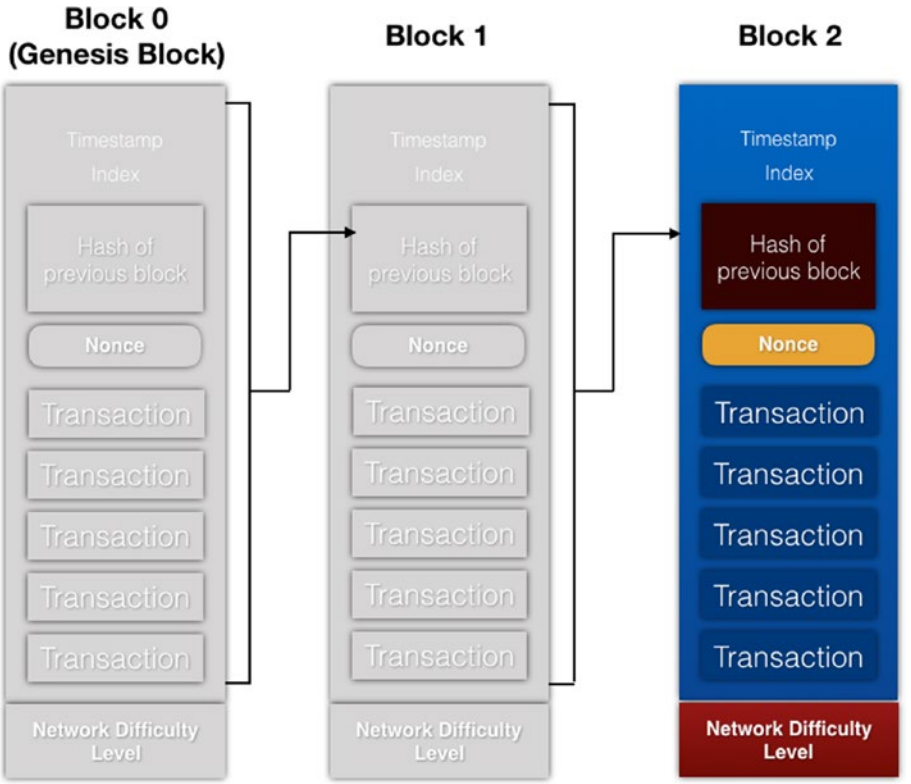

Figure 2-3. *Once a blocked is mined, it will be appended to the blockchain with the timestamp added to the block*

Installing Flask

For our conceptual blockchain, we will run it as a REST API, so that you can interact with it through REST calls. For this, we will use the **Flask** microframework. To install Flask, type the following commands in Terminal:

```
$ pip install flask
$ pip install requests
```

The preceding command installs the Flask microframework.

Tip Flask is a web framework that makes building web applications easy and rapid.

Importing the Various Modules and Libraries

To get started, let's create a text file named **blockchain.py**. At the top of this file, let's import all the necessary libraries and modules:

```
import sys

import hashlib
import json

from time import time
from uuid import uuid4

from flask import Flask, jsonify, request

import requests
from urllib.parse import urlparse
```

Declaring the Class in Python

To represent the blockchain, let's declare a class named `blockchain`, with the following two initial methods:

```
class Blockchain(object):

    difficulty_target = "0000"

    def hash_block(self, block):
        block_encoded = json.dumps(block,
            sort_keys=True).encode()
        return hashlib.sha256(block_encoded).hexdigest()

    def __init__(self):
        # stores all the blocks in the entire blockchain
        self.chain = []

        # temporarily stores the transactions for the
        # current block
        self.current_transactions = []
```

```
        # create the genesis block with a specific fixed hash
        # of previous block genesis block starts with index 0
        genesis_hash = self.hash_block("genesis_block")
        self.append_block(
            hash_of_previous_block = genesis_hash,
            nonce = self.proof_of_work(0, genesis_hash, [])
        )
```

The preceding creates a class named `blockchain` with two methods:

- The `hash_block()` method encodes a block into array of bytes and then hashes it; you need to ensure that the dictionary is sorted, or you'll have inconsistent hashes later on
- The `__init__()` function is the constructor for the class. Here, you store the entire blockchain as a list. Because every blockchain has a genesis block, you need to initialize the genesis block with the hash of the previous block, and in this example, we simply used a fixed string called "genesis_block" to obtain its hash. Once the hash of the previous block is found, we need to find the nonce for the block using the method named `proof_of_work()` (which we will define in the next section).

The `proof_of_work()` method (detailed next) will return a nonce that will result in a hash that matches the difficulty target when the content of the current block is hashed.

For simplicity, we are fixing the `difficulty_target` to a hash result that starts with four zeros ("`0000`").

Tip The source code for our blockchain is shown at the end of this chapter. For the impatient, you may wish to look at the code while we go through the various concepts in this chapter.

Finding the Nonce

We now define the `proof_of_work()` method to find the nonce for the block:

```
    # use PoW to find the nonce for the current block
    def proof_of_work(self, index, hash_of_previous_block,
    transactions):
        # try with nonce = 0
        nonce = 0

        # try hashing the nonce together with the hash of the
        # previous block until it is valid
        while self.valid_proof(index, hash_of_previous_block,
            transactions, nonce) is False:
            nonce += 1

        return nonce
```

The `proof_of_work()` function first starts with zero for the nonce and check if the nonce together with the content of the block produces a hash that matches the difficulty target. If not, it increments the nonce by one and then try again until it finds the correct nonce.

The next method, `valid_proof()`, hashes the content of a block and check to see if the block's hash meets the difficulty target:

```
    def valid_proof(self, index, hash_of_previous_block,
        transactions, nonce):

        # create a string containing the hash of the previous
        # block and the block content, including the nonce
        content =
f'{index}{hash_of_previous_block}{transactions}{nonce}'.encode()
        # hash using sha256
        content_hash = hashlib.sha256(content).hexdigest()

        # check if the hash meets the difficulty target
        return content_hash[:len(self.difficulty_target)] ==
            self.difficulty_target
```

Appending the Block to the Blockchain

Once the nonce for a block has been found, you can now write the method to append the block to the existing blockchain. This is the function of the `append_block()` method:

```
# creates a new block and adds it to the blockchain
def append_block(self, nonce, hash_of_previous_block):
    block = {
        'index': len(self.chain),
        'timestamp': time(),
        'transactions': self.current_transactions,
        'nonce': nonce,
        'hash_of_previous_block': hash_of_previous_block
    }

    # reset the current list of transactions
    self.current_transactions = []

    # add the new block to the blockchain
    self.chain.append(block)
    return block
```

When the block is added to the blockchain, the current timestamp is also added to the block.

Adding Transactions

The next method we will add to the `Blockchain` class is the `add_transaction()` method:

```
def add_transaction(self, sender, recipient, amount):
    self.current_transactions.append({
        'amount': amount,
        'recipient': recipient,
        'sender': sender,
    })
    return self.last_block['index'] + 1
```

This method adds a new transaction to the current list of transactions. It then gets the index of the last block in the blockchain and adds one to it. This new index will be the block that the current transaction will be added to.

To obtain the last block in the blockchain, define a property called `last_block` in the `Blockchain` class:

```
    @property
    def last_block(self):
        # returns the last block in the blockchain
        return self.chain[-1]
```

Exposing the Blockchain Class as a REST API

Our `Blockchain` class is now complete, and so let's now expose it as a REST API using Flask. Append the following statements to the end of the **blockchain.py** file:

```
app = Flask(__name__)

# generate a globally unique address for this node
node_identifier = str(uuid4()).replace('-', '')

# instantiate the Blockchain
blockchain = Blockchain()
```

Obtaining the Full Blockchain

For the REST API, we want to create a route for users to obtain the current blockchain, so append the following statements to the end of **blockchain.py**:

```
# return the entire blockchain
@app.route('/blockchain', methods=['GET'])
def full_chain():
    response = {
        'chain': blockchain.chain,
        'length': len(blockchain.chain),
    }
    return jsonify(response), 200
```

Performing Mining

We also need to create a route to allow miners to mine a block so that it can be added to the blockchain:

```
@app.route('/mine', methods=['GET'])
def mine_block():
    blockchain.add_transaction(
        sender="0",
        recipient=node_identifier,
        amount=1,
    )

    # obtain the hash of last block in the blockchain
    last_block_hash =
        blockchain.hash_block(blockchain.last_block)

    # using PoW, get the nonce for the new block to be added
    # to the blockchain
    index = len(blockchain.chain)
    nonce = blockchain.proof_of_work(index, last_block_hash,
        blockchain.current_transactions)

    # add the new block to the blockchain using the last block
    # hash and the current nonce
    block = blockchain.append_block(nonce, last_block_hash)
    response = {
        'message': "New Block Mined",
        'index': block['index'],
        'hash_of_previous_block':
            block['hash_of_previous_block'],
        'nonce': block['nonce'],
        'transactions': block['transactions'],
    }
    return jsonify(response), 200
```

When a miner managed to mine a block, he must receive a reward for finding the proof. Here, we added a transaction to send one unit of rewards to the miner to signify the rewards for successfully mining the block.

When mining a block, you need to find hash of the previous block and then use it together with the content of the current block to find the nonce for the block. Once the nonce is found, you will append it to the blockchain.

Adding Transactions

Another route that you want to add to the API is the ability to add transactions to the current block:

```
@app.route('/transactions/new', methods=['POST'])
def new_transaction():
    # get the value passed in from the client
    values = request.get_json()

    # check that the required fields are in the POST'ed data
    required_fields = ['sender', 'recipient', 'amount']
    if not all(k in values for k in required_fields):
        return ('Missing fields', 400)

    # create a new transaction
    index = blockchain.add_transaction(
        values['sender'],
        values['recipient'],
        values['amount']
    )

    response = {'message':
        f'Transaction will be added to Block {index}'}
    return (jsonify(response), 201)
```

Examples of transactions are users sending cryptocurrencies from one account to another.

Testing Our Blockchain

We are now ready to test the blockchain. In this final step, add the following statements to the end of **blockchain.py**:

```
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=int(sys.argv[1]))
```

In this implementation, we allow the user to run the API based on the specified port number.

To start the first node, type the following command in Terminal:

```
$ python blockchain.py 5000
```

You will see the following output:

```
* Serving Flask app "blockchain" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

Our first blockchain running on the first node is now running. It is also listening at port 5000, where you can add transactions to it and mine a block.

In another Terminal window, type the following command to view the content of the blockchain running on the node:

```
$ curl http://localhost:5000/blockchain
```

You will see the following output (formatted for clarity):

```
{
    "chain": [{
        "hash_of_previous_block": "181cfa3e85f3c2a7aa9fb74f992d0d061d3e4a6
        d7461792413aab3f97bd3da95",
        "index": 0,
        "nonce": 61093,
        "timestamp": 1560757569.810427,
        "transactions": []
```

```
    }],
    "length": 1
}
```

Note The first block (index 0) is the genesis block.

Let's try mining a block to see how it will affect the blockchain. Type the following command in Terminal:

```
$ curl http://localhost:5000/mine
```

The block that is mined will now be returned:

```
{
    "hash_of_previous_block": "0e8431c4a7fe132503233bc226b1f68c9d2bd4d30af
    24c115bcdad461dda48a0",
    "index": 1,
    "message": "New Block Mined",
    "nonce": 24894,
    "transactions": [{
        "amount": 1,
        "recipient": "084f17b6e5364cde86a231d1cc0c9991",
        "sender": "0"
    }]
}
```

Note Observe that the block contains a single transaction, which is the reward given to the miner.

You can now issue the command to obtain the blockchain from the node:

```
$ curl http://localhost:5000/blockchain
```

You will now see that the newly mined block is in the blockchain:

```
{
    "chain": [{
        "hash_of_previous_block": "181cfa3e85f3c2a7aa9fb74f992d0d061d3e4a6d
        7461792413aab3f97bd3da95",
        "index": 0,
        "nonce": 61093,
        "timestamp": 1560757569.810427,
        "transactions": []
    }, {
        "hash_of_previous_block": "0e8431c4a7fe132503233bc226b1f68c9d2bd4d
        30af24c115bcdad461dda48a0",
        "index": 1,
        "nonce": 24894,
        "timestamp": 1560759370.988651,
        "transactions": [{
            "amount": 1,
            "recipient": "084f17b6e5364cde86a231d1cc0c9991",
            "sender": "0"
        }]
    }],
    "length": 2
}
```

Tip Remember that the default difficulty target is set to four zeros (`difficulty_target = "0000"`). You can change it to five zeros and retest the blockchain. You will realize that it now takes a longer time to mine a block, since it is more difficult to find a nonce that results in a hash beginning with five zeros.

Let's add a transaction to a block by issuing the following command in Terminal:

```
$      curl -X POST -H "Content-Type: application/json" -d '{
"sender": "04d0988bfa799f7d7ef9ab3de97ef481", "recipient":
"cd0f75d2367ad456607647edde665d6f", "amount": 5}' "http://localhost:5000/
transactions/new"
```

Caution Note that Windows does not support single quote (') when using curl in the command line. Hence, you need to use double quote and use the slash character (\) to turn off the meaning of double quotes (") in your double-quoted string. The preceding command in Windows would be

```
curl -X POST -H "Content-Type: application/json" -d "{
\"sender\": \"04d0988bfa799f7d7ef9ab3de97ef481\",
\"recipient\": \"cd0f75d2367ad456607647edde665d6f\",
\"amount\": 5}" "http://localhost:5000/transactions/new"
```

You should see the following result:

```
{"message":"Transaction will be added to Block 2"}
```

You can now mine the block:

```
$ curl http://localhost:5000/mine
```

You should see the following result:

```
{
    "hash_of_previous_block": "282991fe48ec07378da72823e6337e13be8524ced51
    00d55c591ae087631146d",
    "index": 2,
    "message": "New Block Mined",
    "nonce": 61520,
    "transactions": [{
        "amount": 5,
        "recipient": "cd0f75d2367ad456607647edde665d6f",
        "sender": "04d0988bfa799f7d7ef9ab3de97ef481"
    }, {
        "amount": 1,
        "recipient": "084f17b6e5364cde86a231d1cc0c9991",
        "sender": "0"
    }]
}
```

The preceding shows that Block 2 has been mined and it contains two transactions – one that you have added manually and another the rewards for the miner.

You can now examine the content of the blockchain by issuing this command:

```
$ curl http://localhost:5000/blockchain
```

You will now see the newly added block containing the two transactions:

```
{
    "chain": [{
        "hash_of_previous_block": "181cfa3e85f3c2a7aa9fb74f992d0d061d3e4a6
        d7461792413aab3f97bd3da95",
        "index": 0,
        "nonce": 61093,
        "timestamp": 1560757569.810427,
        "transactions": []
    }, {
        "hash_of_previous_block": "0e8431c4a7fe132503233bc226b1f68c9d2bd4d3
        0af24c115bcdad461dda48a0",
        "index": 1,
        "nonce": 24894,
        "timestamp": 1560759370.988651,
        "transactions": [{
            "amount": 1,
            "recipient": "084f17b6e5364cde86a231d1cc0c9991",
            "sender": "0"
        }]
    }, {
        "hash_of_previous_block": "282991fe48ec07378da72823e6337e13be8524ce
        d5100d55c591ae087631146d",
        "index": 2,
        "nonce": 61520,
        "timestamp": 1560760629.10675,
        "transactions": [{
            "amount": 5,
            "recipient": "cd0f75d2367ad456607647edde665d6f",
            "sender": "04d0988bfa799f7d7ef9ab3de97ef481"
        }, {
            "amount": 1,
```

```
            "recipient": "084f17b6e5364cde86a231d1cc0c9991",
            "sender": "0"
        }]
    }],
    "length": 3
}
```

Synchronizing Blockchains

In real life, a blockchain network consists of multiple nodes maintaining copies of the same blockchain. So, there must be a way for the nodes to synchronize so that every single node is referring to the same identical blockchain.

When you use Python to run the **blockchain.py** application, only one node is running. The whole idea of blockchain is decentralization – there should be multiple nodes maintaining the blockchain, not just a single one.

For our example, we shall modify it so that each node can be made aware of neighboring nodes on the network (see Figure 2-4).

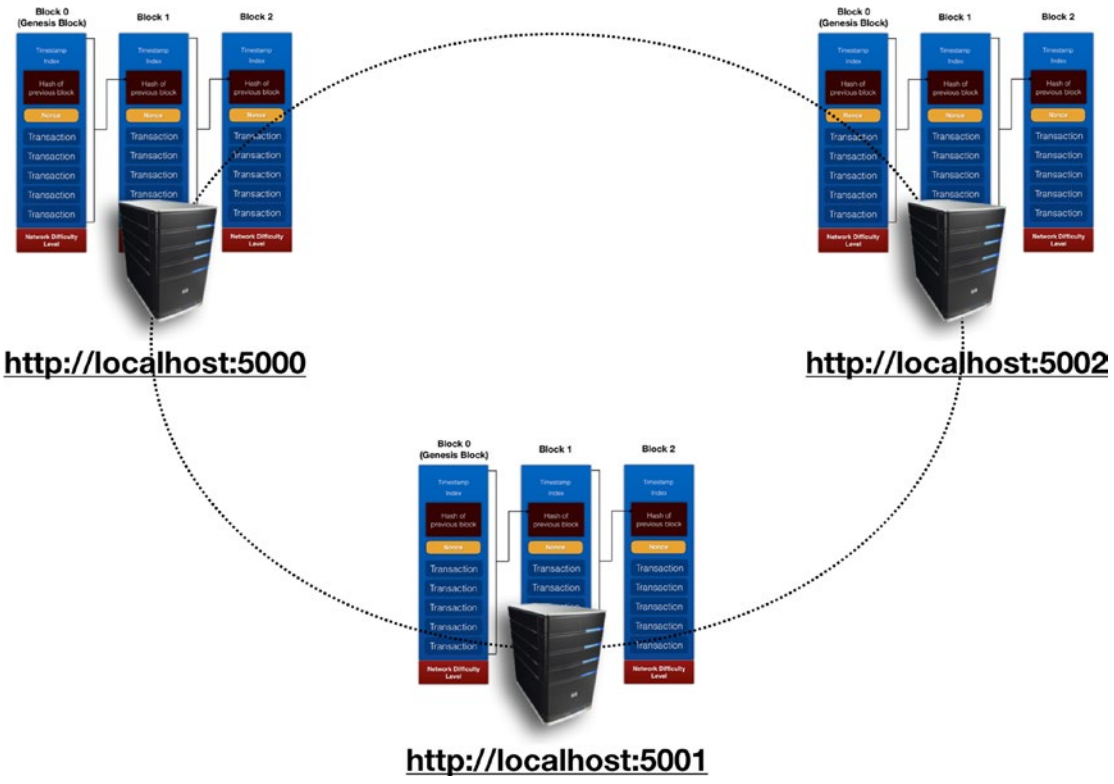

Figure 2-4. *A blockchain network should consist of multiple nodes*

To do that in our example, let's add a number of methods to the `Blockchain` class. First, add a `nodes` member to the constructor of the `Blockchain` class and initialize it to an empty `set`:

```
def __init__(self):
    self.nodes = set()

    # stores all the blocks in the entire blockchain
    self.chain = []
    ...
```

This `nodes` member will store the address of other nodes. Next, add a method called `add_node()` to the `Blockchain` class:

```
def add_node(self, address):
    parsed_url = urlparse(address)
    self.nodes.add(parsed_url.netloc)
    print(parsed_url.netloc)
```

This method allows a new node to be added to the `nodes` member, for example, if "`http://192.168.0.5:5000`" is passed to the method, the IP address and port number "`192.168.0.5:5000`" will be added to the `nodes` member.

The next method to add to the `Blockchain` class is `valid_chain()`:

```
# determine if a given blockchain is valid
def valid_chain(self, chain):

    last_block = chain[0]   # the genesis block
    current_index = 1       # starts with the second block

    while current_index < len(chain):
        block = chain[current_index]
        if block['hash_of_previous_block'] !=
            self.hash_block(last_block):
            return False

        # check for valid nonce
        if not self.valid_proof(
            current_index,
            block['hash_of_previous_block'],
```

```
                block['transactions'],
                block['nonce']):
                return False

            # move on to the next block on the chain
            last_block = block
            current_index += 1

        # the chain is valid
        return True
```

The `valid_chain()` method validates that a given blockchain is valid by performing the following checks:

- It goes through each block in the blockchain and hashes each block and verifies that the hash of each block is correctly recorded in the next block.
- It verifies that the nonce in each block is valid.

Finally, add the `update_blockchain()` method to the `Blockchain` class:

```
    def update_blockchain(self):
        # get the nodes around us that has been registered
        neighbours = self.nodes
        new_chain = None

        # for simplicity, look for chains longer than ours
        max_length = len(self.chain)

        # grab and verify the chains from all the nodes in our
        # network
        for node in neighbours:
            # get the blockchain from the other nodes
            response =
                requests.get(f'http://{node}/blockchain')

            if response.status_code == 200:
                length = response.json()['length']
                chain = response.json()['chain']
```

```
                # check if the length is longer and the chain
                # is valid
                if length > max_length and
                    self.valid_chain(chain):
                    max_length = length
                    new_chain = chain

        # replace our chain if we discovered a new, valid
        # chain longer than ours
        if new_chain:
            self.chain = new_chain
            return True

        return False
```

The `update_blockchain()` method works by

- Checking that the blockchain from neighboring nodes is valid and that the node with the longest valid chain is the authoritative one; if another node with a valid blockchain is longer than the current one, it will replace the current blockchain.

With the methods in the `Blockchain` class defined, you can now define the routes for the REST API:

```
@app.route('/nodes/add_nodes', methods=['POST'])
def add_nodes():
    # get the nodes passed in from the client
    values = request.get_json()
    nodes = values.get('nodes')

    if nodes is None:
        return "Error: Missing node(s) info", 400

    for node in nodes:
        blockchain.add_node(node)

    response = {
        'message': 'New nodes added',
        'nodes': list(blockchain.nodes),
    }
    return jsonify(response), 201
```

The /nodes/add_nodes route allows a node to register one or more neighboring nodes.

The /nodes/sync route allows a node to synchronize its blockchain with its neighboring nodes:

```
@app.route('/nodes/sync, methods=['GET'])
def sync():
    updated = blockchain.update_blockchain()
    if updated:
        response = {
            'message':
              'The blockchain has been updated to the latest',
            'blockchain': blockchain.chain
        }
    else:
        response = {
            'message': 'Our blockchain is the latest',
            'blockchain': blockchain.chain
        }
    return jsonify(response), 200
```

Testing the Blockchain with Multiple Nodes

In the Terminal that is running the **blockchain.py** application, press Ctrl+C to stop the server. Type the following command to restart it:

```
$ python blockchain.py 5000
```

Open another Terminal window. Type the following command:

```
$ python blockchain.py 5001
```

Tip You now have two nodes running – one listening at port 5000 and another at 5001.

Let's mine two blocks in the first node (5000) by typing the following commands in *another* Terminal window:

```
$ curl http://localhost:5000/mine
{
    "hash_of_previous_block": "ac46b1f492997e27612a8b5750e0fe340a217aae89e5
    c0efd56959d87127b4d3",
    "index": 1,
    "message": "New Block Mined",
    "nonce": 92305,
    "transactions": [{
        "amount": 1,
        "recipient": "db9ef69db7764331a6f4f23dbb8acd68",
        "sender": "0"
    }]
}
$ curl http://localhost:5000/mine
{
    "hash_of_previous_block": "790ed48f5d52b3eacd2f419e6fdfb2f6b3142bcfc319
    43e4857b7ba4df48bd98",
    "index": 2,
    "message": "New Block Mined",
    "nonce": 224075,
    "transactions": [{
        "amount": 1,
        "recipient": "db9ef69db7764331a6f4f23dbb8acd68",
        "sender": "0"
    }]
}
```

The first node should now have three blocks:

```
$ curl http://localhost:5000/blockchain
{
    "chain": [{
        "hash_of_previous_block": "181cfa3e85f3c2a7aa9fb74f992d0d061d3e4a6d
        7461792413aab3f97bd3da95",
```

```
        "index": 0,
        "nonce": 61093,
        "timestamp": 1560823108.2946198,
        "transactions": []
    }, {
        "hash_of_previous_block": "ac46b1f492997e27612a8b5750e0fe340a217aae
        89e5c0efd56959d87127b4d3",
        "index": 1,
        "nonce": 92305,
        "timestamp": 1560823210.26095,
        "transactions": [{
            "amount": 1,
            "recipient": "db9ef69db7764331a6f4f23dbb8acd68",
            "sender": "0"
        }]
    }, {
        "hash_of_previous_block": "790ed48f5d52b3eacd2f419e6fdfb2f6b3142bcf
        c31943e4857b7ba4df48bd98",
        "index": 2,
        "nonce": 224075,
        "timestamp": 1560823212.887074,
        "transactions": [{
            "amount": 1,
            "recipient": "db9ef69db7764331a6f4f23dbb8acd68",
            "sender": "0"
        }]
    }],
    "length": 3
}
```


As we have not done any mining on the second node (5001), there is only one block in this node:

```
$ curl http://localhost:5001/blockchain
{
    "chain": [{
```

```
        "hash_of_previous_block": "181cfa3e85f3c2a7aa9fb74f992d0d061d3e4a6d
        7461792413aab3f97bd3da95",
        "index": 0,
        "nonce": 61093,
        "timestamp": 1560823126.898498,
        "transactions": []
    }],
    "length": 1
}
```

To tell the second node that there is a neighbor node, use the following command:

```
$ curl -H "Content-type: application/json" -d '{"nodes" :
["http://127.0.0.1:5000"]}' -X POST  http://localhost:5001/nodes/add_nodes
{
    "message": "New nodes added",
    "nodes": ["127.0.0.1:5000"]
}
```

Tip The preceding command registers a new node with the node at port 5001 that there is a neighboring node listening at port 5000.

To tell the first node that there is a neighbor node, use the following command:

```
$ curl -H "Content-type: application/json" -d '{"nodes" :
["http://127.0.0.1:5001"]}' -X POST  http://localhost:5000/nodes/add_nodes
{
    "message": "New nodes added",
    "nodes": ["127.0.0.1:5001"]
}
```

Tip The preceding command registers a new node with the node at port 5000 that there is a neighboring node listening at port 5001.

Figure 2-5 shows the two nodes aware of each other's existence.

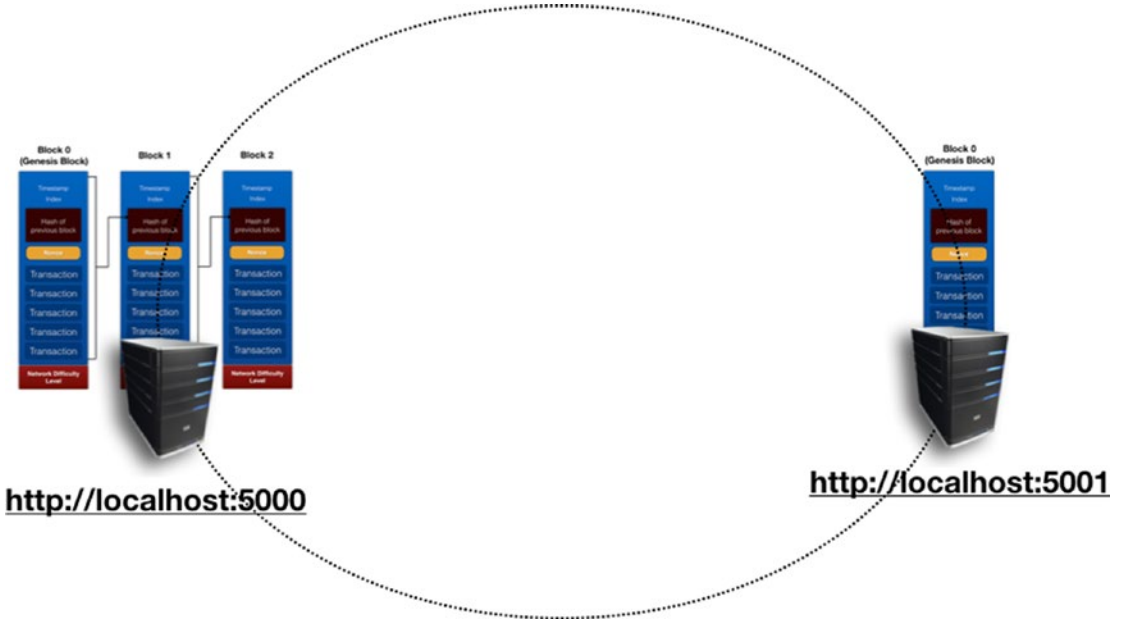

Figure 2-5. *The current states of the two nodes in our blockchain network*

With the first node aware of the existence of the second node (and vice versa), let's try to synchronize the blockchain starting from the first node:

```
$ curl http://localhost:5000/nodes/sync
{
    "blockchain": [{
        "hash_of_previous_block": "181cfa3e85f3c2a7aa9fb74f992d0d061d3e4a6d
        7461792413aab3f97bd3da95",
        "index": 0,
        "nonce": 61093,
        "timestamp": 1560823108.2946198,
        "transactions": []
    }, {
        "hash_of_previous_block": "ac46b1f492997e27612a8b5750e0fe340a217aae
        89e5c0efd56959d87127b4d3",
        "index": 1,
        "nonce": 92305,
        "timestamp": 1560823210.26095,
        "transactions": [{
```

```
            "amount": 1,
            "recipient": "db9ef69db7764331a6f4f23dbb8acd68",
            "sender": "0"
        }]
    }, {
        "hash_of_previous_block": "790ed48f5d52b3eacd2f419e6fdfb2f6b3142bcf
        c31943e4857b7ba4df48bd98",
        "index": 2,
        "nonce": 224075,
        "timestamp": 1560823212.887074,
        "transactions": [{
            "amount": 1,
            "recipient": "db9ef69db7764331a6f4f23dbb8acd68",
            "sender": "0"
        }]
    }],
    "message": "Our blockchain is the latest"
}
```

As the result shows, the first node has the longest chain (three blocks), and hence the blockchain is the latest, and it remains intact. We now synchronize from the second node:

```
$ curl http://localhost:5001/nodes/sync
{
    "blockchain": [{
        "hash_of_previous_block": "181cfa3e85f3c2a7aa9fb74f992d0d061d3e4a6d
        7461792413aab3f97bd3da95",
        "index": 0,
        "nonce": 61093,
        "timestamp": 1560823108.2946198,
        "transactions": []
    }, {
        "hash_of_previous_block": "ac46b1f492997e27612a8b5750e0fe340a217aae
        89e5c0efd56959d87127b4d3",
        "index": 1,
```

```
        "nonce": 92305,
        "timestamp": 1560823210.26095,
        "transactions": [{
            "amount": 1,
            "recipient": "db9ef69db7764331a6f4f23dbb8acd68",
            "sender": "0"
        }]
    }, {
        "hash_of_previous_block": "790ed48f5d52b3eacd2f419e6fdfb2f6b3142bcf
        c31943e4857b7ba4df48bd98",
        "index": 2,
        "nonce": 224075,
        "timestamp": 1560823212.887074,
        "transactions": [{
            "amount": 1,
            "recipient": "db9ef69db7764331a6f4f23dbb8acd68",
            "sender": "0"
        }]
    }],
    "message": "The blockchain has been updated to the latest"
}
```

As the second node's blockchain only has one block, it is therefore deemed outdated. It now replaces its blockchain from that of the first node.

Full Listing for the Python Blockchain Implementation

```
import sys

import hashlib
import json

from time import time
from uuid import uuid4

from flask import Flask, jsonify, request
```

```
import requests
from urllib.parse import urlparse

class Blockchain(object):

    difficulty_target = "0000"

    def hash_block(self, block):
        # encode the block into bytes and then hashes it;
        # ensure that the dictionary is sorted, or you'll
        # have inconsistent hashes
        block_encoded = json.dumps(block,
            sort_keys=True).encode()
        return hashlib.sha256(block_encoded).hexdigest()

    def __init__(self):
        self.nodes = set()

        # stores all the blocks in the entire blockchain
        self.chain = []

        # temporarily stores the transactions for the current
        # block
        self.current_transactions = []

        # create the genesis block with a specific fixed hash
        # of previous block genesis block starts with index 0
        genesis_hash = self.hash_block("genesis_block")
        self.append_block(
            hash_of_previous_block = genesis_hash,
            nonce = self.proof_of_work(0, genesis_hash, [])
        )

    # use PoW to find the nonce for the current block
    def proof_of_work(self, index, hash_of_previous_block,
        transactions):
        # try with nonce = 0
        nonce = 0

        # try hashing the nonce together with the hash of the
```

```
        # previous block until it is valid
        while self.valid_proof(index, hash_of_previous_block,
            transactions, nonce) is False:
            nonce += 1

        return nonce

    # check if the block's hash meets the difficulty target
    def valid_proof(self, index, hash_of_previous_block,
        transactions, nonce):

        # create a string containing the hash of the previous
        # block and the block content, including the nonce
        content =
f'{index}{hash_of_previous_block}{transactions}{nonce}'.encode()

        # hash using sha256
        content_hash = hashlib.sha256(content).hexdigest()

        # check if the hash meets the difficulty target
        return content_hash[:len(self.difficulty_target)] ==
            self.difficulty_target

    # creates a new block and adds it to the blockchain
    def append_block(self, nonce, hash_of_previous_block):
        block = {
            'index': len(self.chain),
            'timestamp': time(),
            'transactions': self.current_transactions,
            'nonce': nonce,
            'hash_of_previous_block': hash_of_previous_block
        }

        # reset the current list of transactions
        self.current_transactions = []

        # add the new block to the blockchain
        self.chain.append(block)
        return block
```

```
    def add_transaction(self, sender, recipient, amount):
        # adds a new transaction to the current list of
        # transactions
        self.current_transactions.append({
            'amount': amount,
            'recipient': recipient,
            'sender': sender,
        })
        # get the index of the last block in the blockchain
        # and add one to it this will be the block that the
        # current transaction will be added to
        return self.last_block['index'] + 1

    @property
    def last_block(self):
        # returns the last block in the blockchain
        return self.chain[-1]

    # --------------------
    # add a new node to the list of nodes e.g.
    # 'http://192.168.0.5:5000'
    def add_node(self, address):
        parsed_url = urlparse(address)
        self.nodes.add(parsed_url.netloc)
        print(parsed_url.netloc)

    # determine if a given blockchain is valid
    def valid_chain(self, chain):

        last_block = chain[0]   # the genesis block
        current_index = 1       # starts with the second block

        while current_index < len(chain):
            # get the current block
            block = chain[current_index]

            # check that the hash of the previous block is
            # correct by hashing the previous block and then
```

```
            # comparing it with the one recorded in the
            # current block
            if block['hash_of_previous_block'] !=
                self.hash_block(last_block):
                return False

            # check that the nonce is correct by hashing the
            # hash of the previous block together with the
            # nonce and see if it matches the target
            if not self.valid_proof(
                current_index,
                block['hash_of_previous_block'],
                block['transactions'],
                block['nonce']):
                return False

            # move on to the next block on the chain
            last_block = block
            current_index += 1

        # the chain is valid
        return True

    def update_blockchain(self):
        # get the nodes around us that has been registered
        neighbours = self.nodes
        new_chain = None

        # for simplicity, look for chains longer than ours
        max_length = len(self.chain)

        # grab and verify the chains from all the nodes in
        # our network
        for node in neighbours:
            # get the blockchain from the other nodes
            response =
                requests.get(f'http://{node}/blockchain')

            if response.status_code == 200:
```

```
                length = response.json()['length']
                chain = response.json()['chain']

                # check if the length is longer and the chain
                # is valid
                if length > max_length and
                    self.valid_chain(chain):
                    max_length = length
                    new_chain = chain

        # replace our chain if we discovered a new, valid
        # chain longer than ours
        if new_chain:
            self.chain = new_chain
            return True

        return False

app = Flask(__name__)

# generate a globally unique address for this node
node_identifier = str(uuid4()).replace('-', '')

# instantiate the Blockchain
blockchain = Blockchain()

# return the entire blockchain
@app.route('/blockchain', methods=['GET'])
def full_chain():
    response = {
        'chain': blockchain.chain,
        'length': len(blockchain.chain),
    }
    return jsonify(response), 200

@app.route('/mine', methods=['GET'])
def mine_block():
    # the miner must receive a reward for finding the proof
```

```
    # the sender is "0" to signify that this node has mined a
    # new coin.
    blockchain.add_transaction(
        sender="0",
        recipient=node_identifier,
        amount=1,
    )

    # obtain the hash of last block in the blockchain
    last_block_hash =
        blockchain.hash_block(blockchain.last_block)

    # using PoW, get the nonce for the new block to be added
    # to the blockchain
    index = len(blockchain.chain)
    nonce = blockchain.proof_of_work(index, last_block_hash,
        blockchain.current_transactions)

    # add the new block to the blockchain using the last block
    # hash and the current nonce
    block = blockchain.append_block(nonce, last_block_hash)
    response = {
        'message': "New Block Mined",
        'index': block['index'],
        'hash_of_previous_block':
            block['hash_of_previous_block'],
        'nonce': block['nonce'],
        'transactions': block['transactions'],
    }
    return jsonify(response), 200

@app.route('/transactions/new', methods=['POST'])
def new_transaction():
    # get the value passed in from the client
    values = request.get_json()

    # check that the required fields are in the POST'ed data
    required_fields = ['sender', 'recipient', 'amount']
```

```
    if not all(k in values for k in required_fields):
        return ('Missing fields', 400)

    # create a new transaction
    index = blockchain.add_transaction(
        values['sender'],
        values['recipient'],
        values['amount']
    )

    response = {'message':
        f'Transaction will be added to Block {index}'}
    return (jsonify(response), 201)

@app.route('/nodes/add_nodes', methods=['POST'])
def add_nodes():
    # get the nodes passed in from the client
    values = request.get_json()
    nodes = values.get('nodes')

    if nodes is None:
        return "Error: Missing node(s) info", 400

    for node in nodes:
        blockchain.add_node(node)

    response = {
        'message': 'New nodes added',
        'nodes': list(blockchain.nodes),
    }
    return jsonify(response), 201

@app.route('/nodes/sync', methods=['GET'])
def sync():
    updated = blockchain.update_blockchain()
    if updated:
        response = {
            'message':
              'The blockchain has been updated to the latest',
```

```
            'blockchain': blockchain.chain
        }
    else:
        response = {
            'message': 'Our blockchain is the latest',
            'blockchain': blockchain.chain
        }
    return jsonify(response), 200

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=int(sys.argv[1]))
```

Summary

In this chapter, you learned how to build your own blockchain using Python. Through this exercise, you learned how

- Blocks are added to the blockchain
- The nonce in a block is found
- To synchronize blockchains between nodes

In the next chapter, you will learn how to connect to the real blockchain – the Ethereum blockchain.