Nicholas Samuel

decentralized applications

[dapps]

An Introduction
for Developers

Decentralized Applications (DAPPS)

An Introduction for Developers

By

Nicholas Samuel

Warning and Disclaimer

Every effort has been made to make this book as accurate as possible. However, no warranty or fitness is implied. The information provided is on an “as-is” basis. The author and the publisher shall have no liability or responsibility to any person or entity with respect to any loss or damages that arise from the information in this book.

Publisher contact

Skinny Bottle Publishing

books@skinnybottle.com

Chapter 1

Understanding Dapps

The Blockchain Technology

The blockchain is the technology underlying decentralized applications, hence the need to familiarize ourselves with the technology. A blockchain is simply a ledger of records which have been organized into blocks, joined together using cryptographic validation. The ledger is not stored in a centralized location and no single entity is tasked with the responsibility of managing the ledger.

With a blockchain, many people can write entries into a record of information, and the users' community is able to control how this record of information is edited and updated. The Wikipedia is a good example of a blockchain. No single publisher is responsible for Wikipedia entries. However, when you descend to the ground level, you find differences which make the blockchain more unique and clear.

The two run on the internet, which is a distributed network. Wikipedia has been built in the World Wide Web on a client-server network model. The Wikipedia entries are stored in a centralized server, and any user whose account has permission can alter these entries. After a user has visited the Wikipedia page, they will be provided with the updated version of "master copy" of the Wikipedia entry. The database is controlled by the Wikipedia administrators, with accesses and permissions on the database being maintained by a central authority.

This scenario is similar to how governments, banks, and insurance companies maintain their centralized databases. The owners are responsible for controlling the centralized databases, including the updates, accesses, and

protection against cyber-threats.

For the case of the blockchain technology, the distributed database created has a different digital backbone. This forms the most distinct feature of the blockchain technology.

For the case of Wikipedia, the master copy is updated on a server and all the users are able to see the new version. In a blockchain, each node in the network comes to the same conclusion, with each node updating the record independently, and the most popular record becomes the de-facto official record.

The blockchain technology presents an innovation for information registration and distribution in which there is no need for a trusted party to facilitate the digital relationships. This makes the blockchain technology very useful.

The blockchain is not a new technology, but it is a combination of three technologies, that is, private key cryptography, internet and a protocol governing incentivization, which have been applied in a new way. This has created a system for digital interactions in which there is no need for a trusted third party. The digital relationships are secured implicitly by the blockchain technology itself.

What is a Decentralized application?

These are applications running on a P2P (Peer-to-Peer) network of computers rather than on a single computer. Decentralized applications have existed since the invention of P2P networks. They are applications which have been developed to run on the internet in such a way that they are not under the control of a single entity.

The following are the characteristics of a Dapp:

1. The application should be open source, operate autonomously and no

single entity should be able to control the application.

2. The app should use a cryptographic token, also known as an App Coin, when accessing the application.

3. All record and data must be cryptographically stored in a decentralized, public blockchain.

4. Tokens must be generated to prove the value nodes contributing to the application.

In Dapps, any change must be decided by a consensus or a majority of its users. The code base for the Dapp should be available for scrutiny. The blockchain validators are rewarded with cryptographic tokens to incentivize them. An agreement must be made on the cryptographic algorithm to be used to show proof of value.

From all the above, we can agree that Blockchain was the first Dapp since it meets all the above requirements. Bitcoin is a blockchain solution implemented to solve some of the problems associated with centralization and censorship. It is a self-sustaining, public ledger which allows efficient transactions with no need for intermediaries and centralized authorities.

Both Bitcoin and Ethereum can be said to be Dapps, but Ethereum is a bigger and better implementation of decentralized applications. The intention of developing Ethereum was to create an alternative protocol for building a decentralized application with a focus on security, development time and scaling. Ethereum can be seen to be the mother of Dapps. It comes with its own language known as Solidity, it helps developers create smart contracts using Ethereum Virtual Machine (EVM). Developers have created Dapps which can be used in real life situations such as resource planning and asset management.

Dapps vs. Smart Contracts

A Dapp is a blockchain-enabled website, in which the smart contract is responsible for allowing it to connect to the blockchain. The best way to know the difference between a Dapp and a smart contract is by understanding how a traditional website functions:

A traditional web application uses CSS, HTML, and JavaScript for rendering a page. When getting details from a database, it uses an API (Application Programming Interface). A good example is Facebook, which uses an API to get your personal details from a database and display them on the page.

For the case of a Dapp, the same technology is used to create and render a page. However, instead of using an API to connect to the database, a smart contract is used to connect the page to the blockchain.

In traditional, centralized applications, the backend code runs on centralized servers, but the backend code in Dapps runs on a decentralized P2P network. Decentralized applications have the whole the package, from the backend to frontend.

The smart contract has only the backend, and it is only a small part of the whole Dapp. If you need to create a decentralized application on a smart contract system, you must combine many smart contracts and use third-party applications to build the front end for the decentralized application.

User Identity in Dapps

One of the good things associated with Dapps is that users can maintain their anonymity. However, there are applications which require the user to verify his identity to be allowed to use the app. Since there is no single authority in Dapps, it becomes hard for the user to verify his identity.

In the case of centralized applications, users can verify their identity by sending some scanned documents, use of OTP verification and so on. This process is known as Know Your User (KYC). In decentralized applications, there is no entity tasked with the responsibility of verifying the identity of the

users, so the Dapp must do this itself.

Dapps are unable to send SMSes or verify scanned documents, meaning that there is a need for these to be provided with digital identities which they can verify.

A digital certificate is amongst the many ways of achieving this. A digital ID can be created and used a digital watermark, in which it will be assigned to each online transaction of any asset.

A digital certificate is sometimes known as a public key certificate. It is just an electronic document which one can use to prove ownership of a public key. The user is expected to have a public key, a private key, and a digital certificate. The user should keep the private key as a secret. He is free to share the public key with anyone. The digital certificate defines the individual who owns the public key, and it is responsible for holding the public key. Normally, a digital certificate can easily be produced by an authorized entity whom you are able to trust. The certificate authority uses his private key to encrypt a field of the digital certificate. This field must then be decrypted by use of the public key of the same certificate authority. If this runs successfully, the authenticity of the digital certificate will have been proved, and it can be said to be valid.

In decentralized applications, the identity of the user can also be proved manually. This can be done by an authorized individual from the company which is providing the client. A good example is when one is creating a Bitcoin account. No one expects the person to prove their identity, but any time they need to withdraw bitcoins to a fiat currency, a proof of identity will be needed. Unverified users will be omitted, and they will not be allowed to use the client. Users who have successfully verified their identity can be allowed to use the client.

With a robust identity management system, the users for the individual systems need to be managed. If different permissions are to be granted to the users, they must determine the kind of permissions each user should be granted.

User Accounts in Decentralized Applications

Centralized applications rely on the use of user accounts. In this case, the data for an account should only be changed by the account owner. For the case of decentralized applications, it is hard for the username-password account policy to be used since passwords cannot be used to show that a particular account change has been requested by the owner.

In decentralized applications, a public-private key pair can be used to represent an account. The hash of a public key can be used to identify the account uniquely. If a user needs to make a change to the data, then they must sign the change using their private key. However, each user of the decentralized application will have to store his or her private key safely. If any user loses his or her private key, they will have lost access to their account permanently.

How to Access Centralized Apps

In some cases, the Dapp may need to access data from a centralized application. The problem in this situation is, how will the Dapp know that the data it gets is the actual response and it has not been altered by a middleman? This problem can be solved through many ways, and this will be determined by the architecture of the Dapp. A good example is how the Ethereum smart contracts access the centralized APIs. In this case, the Oracle service can be used as a middleman. This is because the smart contract is unable to make direct HTTP requests. The Oracle can provide a TLSNotary proof to the smart contract regarding the data it has fetched from the centralized application.

Internal Currency

As an owner of a centralized application, you should win some profit so that

you can have your application running for a longer time. For the case of decentralized applications, there exists no owner. However, for the Dapp to keep on running, it needs some hardware and other network resources. For the Dapp to keep on running, the nodes should be given something in return. This is where there is a need for the internal currency. Most of the Dapps come with a built-in internal currency.

A consensus protocol is used when there is a need to determine the coinsAmount of currency that a node should receive. Based on the consensus protocol being used, only a number of nodes will receive the currency. The nodes responsible for securing the Dapp and keeping it running are the ones which earn currency. Any node which only reads the data will not earn any currency. A good example is in the Bitcoin blockchain, in which only the miners earn bitcoins for mining blocks successfully.

The internal currency is also used to fund the development of decentralized applications, in which they are sold in the form of a token sale. In the Ethereum network, Ether is used as the internal currency. It provides a mechanism for exchange over the network. Note that the price of internal currency in decentralized applications keeps on fluctuating from time to time.

In Ethereum, every smart contract and transaction is expected to be processed independently by all the network nodes. This is a good effort in a bid to ensure that the entire network remains decentralized. This is less efficient when compared to a centralized computer. A trade-off exists between efficiency and decentralization, and this will limit the Ethereum use cases in the future. However, other platforms are being developed with a great focus on scalability rather than scalability.

Permissioned Decentralized Applications

As you might have discovered, Dapps don’t have, meaning that anyone can participate in Dapps without the need to establish an identity.

However, for the case of permissioned Dapps, they are not open for anyone to participate. The permissioned Dapps have all the characteristics of permissionless Dapps, only that you must have permission to participate in them. Different permission systems are used in different permissioned Dapps.

For you to be allowed to join a permissioned Dapp, you must have permission. This means that consensus protocols used in the permissionless Dapps may not apply in the permissioned Dapps. Different consensus protocols are used in permissioned Dapps. Another characteristic of permissioned Dapps is that they don't have an internal currency.

When we consider what happens in Bitcoin, there are no permissions. Anyone can use the cryptographic keys for Bitcoin, and anyone can join the network and act as a node. Anyone can also begin to mine blocks and seek rewards. The miners can also leave whenever they want and return anytime they need to do so. They will get a full record of all the network activities which took place since when they left.

Anyone can read the chain, make changes and write some new block into the chain, provided they adhere to the available rules. This is why Bitcoin is known as a public blockchain. However, this is not the only mechanism one can follow when building a blockchain.

It is possible for blockchains to be developed which require permission for one to be allowed to read information on the network, limit the number of parties able to transact on the blockchain and determine the individuals allowed to write new blocks to the blockchain. A good example of a permissioned blockchain is Ripple. The user to act as the transaction validator on the network must be determined. In Ripple, Microsoft, MIT, and CGI have been added to act as the validators.

With permissioned blockchains, there may be no need for proof of work system or any other requirements from the blockchain nodes. A blockchain developer may want to make it possible for anyone to read the records, but they may not need to make everyone a node.

Chapter 2

Building Decentralized Applications

Decentralized applications are server-less applications capable of running jointly on the client side and on a distributed network which is based on a blockchain like Ethereum. The client device is responsible for managing the user credentials and the front end, while the back-end usually runs on a distributed network of computers, with the computers being responsible for the provision of processing and storage requirements. The business logic is stored in the distributed blockchain network, which gives a clear auditability and transparency of the interactions generated during the interaction of the client and Dapp.

The storage of data is done in decentralized file systems like Swarm or Inter-Planetary File Systems (IPFS). The new model can provide numerous features which can help a wide range of applications.

Dapps have a stronger security. The GUI and user credential management are pushed to the client side, which prevents data breaches from occurring on the server side. The Dapp can store the public key of the user for the purpose of authenticating him, and the signatures re-verified using a challenge-response protocol. The private key is stored on the client device in a secure user wallet, and it is encrypted using a user-supplied password. Data has to be encrypted before it can be stored in the distributed cloud-based storage, swarm, IPFS and other storage alternatives based on cryptocurrency like Storj. Since the user-credentials and data are encrypted and distributed, creating a challenging and sparse attack surface. Any hacker or attacker will have to breach every user individually.

The Dapps run partly on the client side and partly on whole distributed blockchain network, which is made up of thousands of machines. The Dapp

data and logic is replicated, in the form of a smart contract, across several nodes participating in a blockchain network. Distributed operation and replication provide the reliability and availability of the Dapp.

Since Dapps are decentralized by nature, they are resistant to censorship. This is because the smart contract can only be shut down by the original entity which deployed it, or any individual with the private key. Also, when the user interface is decoupled from the business logic, this allows for the creation of alternative interfaces for a similar app, which is a good way of encouraging innovation.

The Architecture of Decentralized Applications

A decentralized application is made up of several nodes which are connected through a blockchain network. Smart contracts run on the blockchain. A user uses a web browser to access the smart contract on the blockchain. This means that the browser runs on the client side, while the blockchain plays the role of a server.

The blockchain is not central, but it is a copy which replicates all the nodes. Note that the client using the browser must not call a smart contract, but it can call anything running on the blockchain.

Data Storage and Retrieval

The following are the requirements for the ideal decentralized database that will meet the needs of a decentralized application:

1. Decentralization and distribution

The applications are decentralization, so the database should also be decentralized.

2. Publicity- the blockchain should everyone to participate in the network, and the database should support this too.

3. High speed- the applications may need to support millions of transactions per second, so that database should be capable of doing this.

4. Sharding support- if the application is expected to be larger and support huge amounts of data, then the total network power should be used to increase the maximum storage capacity. With full replication of data at each node, the risk of losing data is reduced at the nodes as problems may arise with the nodes. However, if the network is too large and is made up of many nodes, replicating data at each node will be too redundant. The replication level may be reduced in favor of increasing the total storage capacity. This means if we are having N nodes, then each record should be replicated to m nodes, where m<N.

5. Ability to delete data- the database should be able to delete the records which are not needed anymore by the applications so that more space can be created.

6. Storage of structured data- the database must have the ability to understand the internal structure of stored data for applications to be allowed to create links between various records.

7. Fault tolerance- the database should tolerate attacks on the network by a continuous provision of services to the users.

8. Secondary indexes- applications should be able to perform fast searches of the records while considering their internal structure.

Let us discuss some of the existing storage technologies and evaluate whether they meet all the above requirements:

IPFS

Interplanetary File System (IPFS) is a distributed file system technology

which works based on Distributed Hash Table (DHT) and the BitTorrent protocol. With this technology, one can combine file systems which are located on different devices. These file systems can be combined into one by use of content addressing.

The following are the advantages associated with the use of IPFS for storage:

1. Each device keeps only the records that it needs, as well as meta-information which describes the location of the files on the other devices. No need for additional motivation for storage of the files.

2. File addressing is done by content, so there is no need to trust the peers.

3. Higher bandwidth due to the use of BitTorrent.

4. It is hard to upload unnecessary files to the network (flooding) since the files can only be placed in the originating device.

The following are the disadvantages associated with the use of IPFS for storage:

1. It only stores files, that is, unstructured information.

2. Once the file has been placed, one cannot leave the network until when the file can be downloaded by another person.

3. The files cannot be changed, that is, they are static.

4. There is no guarantee for data storage by other devices.

Data File Storage

These are storages which provide one with a way to merge several storage devices into one cloud storage device. The cloud storage then allows the users to store their files and data in the same way they could have done in a centralized storage such as Dropbox.

The device users, commonly known as “farmers” provides a place where other users’ files can be stored. The users, in turn, pay them money according to their contribution. For you to measure the contribution accurately or stop abuse and ensure the storage is reliable, there are several measures which you can use, including proof of retrievability, proof of storage, which work based on cryptography. For a successful verification passing, the user must pay, and the farmer must receive a certain coinsAmount in the form of cryptocurrency.

Most of these projects are build using content addressing and DHT technology. Others additionally use smart contracts and blockchain. Currently, there are several projects in the market including Storj, Sia, MadeSAFE and Ethereum Swarm. These technologies are build based on similar principles. The Ethereum Swarm was developed to provide some convenient file storage for decentralized applications.

The following are the advantages of using distributed file storage:

1. The storage of the files is done in the cloud, meaning that they will constantly be available regardless of the availability of the owner.

2. They have a higher throughput.

3. It is possible for one to delete any unnecessary files.

4. There is a financial motivation, ensuring there is a reliability in storage and retrieval of the files.

The following are the disadvantages of using distributed file storage:

1. It can only be used for storage of files but not structured information.

2. The storage space is not given for free.

3. The files are static.

4. Distributed file repositories are attractive for storage of files. However, storage of structured dynamic information in static files is a big problem. This is because the file repositories do not know anything regarding the file

contents. This information is very important in our case for searching for information based on the name (or identifier) or by the contents.

Distributed Databases

Creation of a fully distributed database capable of fully supporting availability, consistency and petition tolerance is impossible.

When developing Dapps, there for a distributed database, and this database should be accessible and partition-tolerant. This restricts our selection to only the NoSQL databases.

There are various distributed NoSQL databases, including Cassandra, MongoDB, and RethinkDB. These are able to work with a large number of replicas which are clustered together. The client normally has one of these and the data is synchronized to the rest of the replicas. We can use sharding for the purpose of load balancing in cases where the data on part of the replicas only. Once new replicas are added to the cluster, the cluster scales linearly. In some implementations, the replica is allowed to take part of cluster load automatically.

With NoSQL databases, we can achieve “eventual consistency”, in that the data will be matched after some time once the individual replicas have been synchronized.

The following are the advantages associated with the use of distributed applications for storage in Dapps:

1. They scale linearly in terms of speed and storage capacity.

2. Mature realizations.

3. High speed.

4. Resistant to inaccessibility of individual replicas.

The following are the disadvantages associated with using distributed databases:

1. Peers should be trusted.

BigChainDB

This is also known as InterPlanetary Database. It has a very high transaction speed of about 1 million transactions per second. It uses a distributed storage with partial replication, which gives it a huge storage capacity. It relies on a very consensus when building the blocks, and all blocks and transactions are stored in a NoSQL database such as MongoDB or RethinkDB.

Each node in the architecture is granted full rights so as to write to a common data store. This indicates that the system is not fault tolerant. This problem has limited the use of BigChainDB to private networks.

It has the following advantage:

1. It has a high speed and storage capacity when compared to the distributed NoSQL databases.

The following are the disadvantages of BigChainDB:

1. Immutability, meaning that the data cannot be deleted legally, but it can be deleted maliciously.

2. Byzantine fault tolerance, which makes it unsuitable for use in public networks.

For a database to be linked to decentralized applications, it must be resistant to the malicious behavior of the other database nodes, provide adequate coinsAmount of replication and be able to motivate the participants to support the network. For a database to be suitable for use in decentralized applications, it should have the following characteristics:

1. Every user should be able to read all records.

2. The database should be public, the database user should be identified by a public key, normally the user ID.

3. Each user should be able to run transactions on the database, and the user must sign each transaction.

4. Only the record owner should be able to implement the changes or a user who is trusted via the permission mechanism being used.

5. All record keys should have the user ID as their prefix. This will ensure that record keys for different users don't conflict.

6. The blockchain smart contracts can be used to configure more complex permissions, example, rights for creation and deletion of tables, trust between specific users etc.

7. Once a user creates a new record, it imprints ownership of the data.

8. Permissions should be checked for both replication and transactions.

The purpose of the mandatory cryptographic signature is to ensure that its deletion or modification by a malicious node is impossible without knowing the private key of its owner. With this, the database will be Byzantine fault tolerant even in the absence of a consensus protocol.

Our First Dapp

A decentralized application is made up of two parts, the front end, and the backend. The front end is written in HTML, while the backend is the database. The front-end part has full network access, so you can continue using Bootstrap or any other framework that you like using. The process of developing the front end for a Dapp is similar to the development of a website. Reactive programming can be used by using the callback functions, meaning there is no need for you to learn a new framework.

In this chapter, we will be using Alethzero as the development client to create a simple coin contract, which will form the backend of our application.

Setting up the Environment

Alethzero is the C++ implementation of Ethereum and it has been designed to be used by developers. We will install the "master" version of Alethzero due to its stability and the fact that it has all the latest features.

Download the binaries for Alethzero based on the operating system you are using.

Next, you should install MIX. This will be used as the integrated development environment (IDE). You should finally install Mist, which will provide you with an environment for testing your Dapp and fine-tuning the front end during the development process.

Once you start Alethzero, you will be presented with several interfaces. Your resolution may vary, and you may not see all the interfaces. Just click the "X" so as to close all the panes manually, resize the screen to fit your resolution, then read the interfaces manually by right-clicking below the title bar and the right of "refresh" button.

At the center of the screen, you should see a browser window, forming the Webkit view. The existing web can be browsed from there. You can try something like Facebook.

The other panels have some technical and debug information. This interface may not be much friendly to users, but it is very useful to the developers. The final Ethereum browser named "Mist" and build on top of Go Ethereum implementation will provide a different look and feel.

Note that the Ethereum platform is generalized, and it can be used for building financial applications, games, social networks, gambling applications and other types of applications.

We will write a simple contract which functions like a bank. However, this will have a transparent ledger which can be viewed by the whole world. We will issue a number of tokens, then send them to friends. In web 2 world, this app would be implemented in PHP and MySQL, but the users should trust you as an honest individual, that you always keep a consistent ledger, hackers cannot break into your server and that you have honest employees who cannot plant a backdoor.

The Contract

The backend, which is referred to as "contract" in Ethereum will use a language named Solidity. There are other contract languages which can be used when developing the backend in Ethereum. Examples of these include Serpent which is similar to Python and LLL which is similar to Lisp. Solidity is officially supported by ETHDEV team; hence we will be using it. Since we are building a bank, we will have to do the following:

1. Create an account with some tokens so that the contract can run smoothly once it is started.

2. Create a "send" function which is similar to "contract.send(account, coinsAmount)" so as to move the tokens around.

Open the text editor of your choice then write the contract as follows:

```
contract metaCoinContract {

        mapping (address => uint) balances;

        function metaCoinContract() {

                balances[msg.sender] = 10000;

        }

        function sendCoinFunction(address coinsReceiver, uint
coinsAmount) returns(bool enough) {

                if (balances[msg.sender] < coinsAmount) return
```

```
false;
                balances[msg.sender] -= coinsAmount;
                balances[coinsReceiver] += coinsAmount;
                return true;
        }
}
```

A contract is divided into methods. We have created the first method and named it “metaCoinContract”. It is a contractor method defining the initial state of the data storage for our contract. Constructor functions should have the same name as the contract. Note that the initialization code is only executed once during the creation of the contract, and it is never executed again. The account has 10,000 tokens as stated in the code.

We then have the contract code, which is immutable, and backed by millions of nodes which will ensure that it returns the right results every time. This is the function which checks whether the sender has enough balance and if so, it will transfer some tokens from one account to another. Consider the following line which has been extracted from our previous code:

mapping (address => uint) balances;

The line creates a mapping in the storage where the code can write information to the contract’s storage. We have defined a mapping of key-value pairs of type uint and address named “balances”. This is where the users’ coin balances will be stored.

Also, two data types have been specified, that is, address and uint. Solidity is a statistically typed language, meaning that type checking is done during the compile time. This is not the case with Serpent. When the type is specified before compile time, the size of the array to be passed by the transactions will be reduced, and the compiler will be in a position to create a more optimized code:

```
function metaCoinContract() {
```

```
                balances[msg.sender] = 10000;
}
```

The contract has been initialized, and it will only be executed once since it has been created within the constructor method named “metaCoinContract”. It does many things. First, it will use msg.sender to look for the public address of transaction sender, who is you in this case. Secondly, it uses the mapping balances to access the contracts storage. In contracts, data is stored in the form of key-value pairs both having 32 bytes in length.

The msg.sender denotes the public key and it is a number of 160 bits. This identifier is unique in the network and it can’t be forged due to the cryptographic laws used in Ethereum network.

Now that we have defined our balance, we can explore the “sendCoinFunction” method which will be executed every time the contract is called from now. This forms the only executable function that our user can call. We can’t call the initialization function again:

```
function sendCoinFunction(address coinsReceiver, uint
coinsAmount) returns(bool enough) {

                if (balances[msg.sender] < coinsAmount) return
false;

                balances[msg.sender] -= coinsAmount;

                balances[coinsReceiver] += coinsAmount;

                return true;

        }
```

The transaction is passing two arguments to the function. The “coinsReceiver” is a public address for the recipient and it takes 160 bits. The “coinsAmount” denotes the number of tokens that you need to send to the coinsReceiver.

In the first line, we are checking whether the balance for “msg.sender” is less than what we need to send. If this is the case, we will return false and the next two lines of code will not be executed. The coinsAmount which calls the

contract is the one which sends tokens, hence this makes sense, and there must be sufficient tokens for the transaction to be completed.

Three arguments have been passed to the function “transfer()” which is responsible for sending tokens from one storage address to another. These arguments include “msg.sender”, “coinsReceiver” and “coinsAmount”.

In case the balance checks out, our conditional will evaluate to a false. The next two lines will then subtract the coinsAmount to be sent from sender balance:

```
balances[msg.sender] -= coinsAmount;
```

This will then be added to the balance of the account which is to receive the tokens:

```
balances[coinsReceiver] += coinsAmount;
```

At this point, we are having a function responsible for sending tokens from one account to another.

The Ethereum Gas

The “gas” is an important concept which will help you to master the Ethereum smart contracts. The decentralized web should be powered to keep on running. Ethereum cannot work under the reliance of any centralized authority. This is because the authority may end up manipulating the database. Instead of this, each node in the network holds a copy of the decentralized database and they can audit it.

The nodes of the network process the code which is being executed in the database and they vote to come to an agreement regarding the correct state of the database. The majority vote will win, and the nodes are incentivized to do the verification. The voting is normally done at regular intervals, most after every 12.7 seconds. The contract we have written will be stored in the database. The contract will be triggered then executed once other users or the

other contracts call it.

This approach has a limitation in terms of the processing speed. The total processing power of the Ethereum network regardless of the total number of nodes forming it is small. This is why you should not store megabytes of data in the Ethereum network or render your 3d graphics. However, new mechanisms for doing this are under implementation.

Also, due to the limited computation power, the right measurement should be done so that a single actor doesn't commit evil deeds like running infinite loops on the Ethereum nodes. This measurement is done using a unit called "gas".

For a function to be executed, gas is needed. When calling a function, you should specify the coinsAmount of gas that you will need to send to the contract, and the coinsAmount you need to pay for the gas, which is priced in ether.

The various operations that a contract can support are normally priced differently. Example, a single execution can cost only one gas. Others such as writing to the storage normally cost more because the storage is a scarce resource. In case more than enough gas is sent to a contract and it doesn't use all of it, then you will be refunded the same. If not enough gas is sent to the contract, the contract will stop and roll back. The pricing of the gas is determined by the global consensus of the community. This is an indication that the operations whose gas is well priced will be executed first, then the rest will be executed later.

Uploading the Contract

Now that we have written our first Ethereum contract, we should upload it to the Ethereum network.

Begin by launching the Alethzero client and familiarize yourself with its interface. Ensure that you have opened a notepad with the contract written inside of it. Don't connect to Testnet, but choose "Use Private chain" from

debug menu then create your private chain. With this, it will not be a must for the testnet to be online when you are developing the Dapps.

Click the button written “New Transaction” from the menu bar to open the dialog for the new transaction. This is where you will add the code for the contract and send transactions. It consists of the address we are sending the transactions to, the options for gas and gasPrice, a pane for inputting the code for the contract or the transaction data. You can close it for now.

For us to save or operate the contract, ether will be required. For now, we have 0 units of ether. For us to acquire ether, we must participate in the voting process which will guarantee the integrity of a decentralized database, and this process is known as “mining”.

At this point, we are running a private chain instead of connecting to the network, meaning that we can only mine for new blocks. From the toolbar, click “Mine”, go to Debug then choose “Force Mining”. Most tabs should come alive with information. The mining tab provides a visual representation of mining of a block, and once a successful block is found, a red spike will mark it and your balance will increase. You should look at the “Blockchain” tab as each vote will be recorded there then tagged with a number.

Once you have reached 15000 Finney’s, you can choose to stop mining. To stop the mining process, just click “Stop”. It is time for us to deploy the contract.

Reopen the popup for “New Transaction” then copy then paste the solidity code we had written earlier in the “Data” textbox. Our intention is to create a contract then send ether to some other account. You can choose to leave all the available fields with their default values. We will send 10,000 ether to a contract priced 10 Szabo each. The Szabo is a unit of value, which is equal to 0.000001 ether. The code for creation of the contract is very simple, and most of this ether will be refunded to us as all the gas cannot be consumed by saving of the contract.

If the contract was pasted correctly, two messages should be seen in the pane under the contract code. The first one will have the title “Solidity” and it should look like a JavaScript snippet, a list of executable functions contained

in the contract, and the ABI (Application Byte Interface) for the function "sendCoinFunction". Copy the information and paste it into Notepad as it will be needed later. The second one will be a series of assembly-like instructions which are similar to "PUSH2 0x27 0x10 CALLER…" This forms the EVM code for your contract in a compiled form.

At this point, our contract has been compiled, so we should deploy it to the decentralized database by clicking "Execute". Since you should not be mining, the contract will be "Pending". Ensure the "Pending" pane is open then observe what you see on it.

Once you have pressed the "Execute" on transact pane, make some quick note of the "creates" field for the transaction in "pending" pane. In this case, it should be "1f530b6b…"

The contract is now pending, and you should commit it to the blockchain simply by mining some block. Click mine and wait for the transaction to disappear from pending pane and for some new entry to appear in "contract" pane. Turn off the mining again then click the new contract in the contract pane.

You will see the entry in the contracts data storage from the initial setup showing a key-value pair of SHA3 of the public address and number 10,000. We only must ensure that the constructor function we have functions in the right way. It is time for us to test whether tokens can be sent to some other public key. For us to run the "sendCoinFunction" function, we should send a transaction to the address of the contract and specify the "To" address, a "value" and a function ID in transactions data array. Find the transact pane then paste the address of the contracts in the "Address" box.

In the section where we added our code, we will first add the function id and the recipient address and the value on separate lines. When the contract was earlier sent to the network, the function ID "sendCoinFunction" should have been stored. Use your public address on testnet and any value below 10,000. Click "Execute" and you will see the transaction pending, click "mine" and you will see a new bloc generated then stop mining.

You will have successfully created your first contract. You can add the

contract to a test network if you wish.

Chapter 3

Typing and Storage

Solidity supports seven basic data types. These include the following:

- hash- a 256-bit, 32-byte data chunk. It can be indexed into bytes and operated with bitwise operations.

- uint- a 256-bit unsigned integer. It can be operated with unsigned and bitwise arithmetic operations.

- int- a 256-bit signed integer. It can be operated with signed and bitwise arithmetic operations.

- string32- a zero-terminated ASCII string with a maximum length of 32-bytes (256-bit).

- address- an account identifier, similar to 160-bit hash type.

- bool- a two-state value.

Solidity is a statically typed language, meaning that variable types should be known during compile time. This is not the same as JavaScript which supports loose typing which facilitates more flexibility in variable typing. Explicit type conversion is only allowed during conversion between the signed and unsigned integers or during the conversion of an integer or address to a hash.

One can specify the size of a uint, hash or int by simply suffixing the required bit width in 8-bit steps. Other than these types, it is possible for one to define mappings and structs as datatypes. Every type should be based on the 32-bit word limit of Ethereum.

Contracts Storage

Each blockchain contract has a storage to which it's the only one allowed to write to. This is referred to as contracts state and it is a flexible database which provides one with almost an unlimited storage space if you are ready to pay for it.

A contract's storage is in its most basic key-value store with up to 2^256 keys and 2^256 values. This will give you a sufficient storage for the creation of any possible database storage structures you can imagine.

Solidity provides its users with various tools which can be used for structuring the contract's storage into a relational database model. In our previous contract, we used a mapping for storage of the value associated with the address, but this is just one of the many available alternatives.

Structs

For C programmers, you must be familiar with structs. They are simply a physical grouping of variables which are stored under one reference. Consider the following example:

```
struct coinsWalletStruct {

        uint aRedCoin;

        uint aGreenCoin;

}
```

In the above code, we have defined a type of struct, but we have not initialized its instance. This means that once a struct has been defined, nothing is written to the contracts storage. To do this, at least a single instance of the struct should be defined and a value assigned to any one of its

parameters. Example:

```
coinsWalletStruct wallet1;

wallet1.aRedCoin = 500

wallet1.aGreenCoin = 250

With the above, the addresses 0 and 1 for the storage will be
written to respectively. We can then go ahead and deploy the
contract in Alethzero:

contract Test{

        struct coinsWalletStruct {

        uint aRedCoin;

        uint aGreenCoin;

        }

        coinsWalletStruct wallet1;

        function Test(){

                wallet1.aRedCoin = 500;

                wallet1.aGreenCoin = 250;

        }

}
```

In the above example, we are using the dot operator on the wallet1 instance of coinsWalletStruct to access the underlying parameters of the wallet1 variable. When it is used as a variable of finite size, that is, not inside a mapping, structs will place things continuously in the storage while beginning at key value “0*0”. You have deployed it in Alethzero, ensure you look inside contract pane to observe the storage addresses it writes to.

Mappings

Structs become even more interesting when they are used as data types for values inside mappings. Consider the metaCoinContract contract given below which is amended for each user to have two balances:

```
contract metaCoinContract {

        struct coinsWalletStruct {

        uint aRedCoin;

        uint aGreenCoin;

        }

        mapping (address => coinsWalletStruct) balances;

        function metaCoinContract() {

                balances[msg.sender].aRedCoin = 10000;

                balances[msg.sender].aGreenCoin = 5000;

        }

        function sendRed(address coinsReceiver, uint coinsAmount)
returns(bool successful) {

                if (balances[msg.sender].aRedCoin < coinsAmount)
return false;

                balances[msg.sender].aRedCoin -= coinsAmount;

                balances[coinsReceiver].aRedCoin += coinsAmount;

                return true;

        }

        function sendGreen(address coinsReceiver, uint
coinsAmount) returns(bool successful) {

                if (balances[msg.sender].aGreenCoin <
coinsAmount) return false;

                balances[msg.sender].aGreenCoin -= coinsAmount;

                balances[coinsReceiver].aGreenCoin +=
coinsAmount;
```

```
                return true;

        }

}
```

Mappings normally calculate the references for a data storage simply by hashing the keys which are passed to the mapping. Mappings can also be included as data types in another mapping. This is demonstrated below:

```
contract rainbowCoinContract {

        mapping (address => mapping (uint => uint)) balances;

        function rainbowCoinContract() {

                balances[msg.sender][0] = 10000; ///a red coin

                balances[msg.sender][1] = 10000; /// a orange
coin

                balances[msg.sender][2] = 10000; /// a yellow
coin

                balances[msg.sender][3] = 10000; /// a green coin

                balances[msg.sender][4] = 10000; /// a blue coin

                balances[msg.sender][5] = 10000; /// a indigo
coin

                balances[msg.sender][6] = 10000; /// a violet
coin

        }

        function sendCoinFunction(address coinsReceiver, uint
coinsAmount, uint coin) returns(bool successful) {

                if (balances[msg.sender][coin] < coinsAmount)
return false;

                balances[msg.sender][coin] -= coinsAmount;

                balances[coinsReceiver][coin] += coinsAmount;

                return true;

        }
```

```
}
```

This allows us to access elements in sub mapping inside the balances by use of square brackets [][].

Mappings can also be added within structs, and place structs inside mappings. Note that you can place any data type inside a struct other than the struct or the mapping itself. You can now deploy the above contract, that is, rainbowContract, simply by pasting it into Alethzero transact window. You will note that it behaves simply like our first contract. However, when trying to send a transaction, you will have to specify a third argument which is "coin". This should be an integer whose value ranges between 0 and 6.

Accessor Functions

You know how to view the contents of a contract added to the network by use of Alethzero's GUI. However, we have not defined any way that you can directly use to get the value associated with a key. This is a requirement for most contracts because as external actors, we can view the contracts storage easily. However, other contracts are not able to look at the other contracts storage without execution of a message call to a function which has been specifically defined to return a relevant information.

Solidity uses the "public" keyword to create an accessor function to any state variables we have chosen to make public. This is done automatically. We can modify the rainbowContract so that we can make the Solidity automatically create an accessor function. This is demonstrated below:

```
contract rainbowCoinContract {

        mapping (address => mapping (uint => uint)) public
balances;

        function rainbowCoinContract() {

                balances[msg.sender][0] = 10000; /// a red coin

                balances[msg.sender][1] = 10000; /// a orange
```

```
coin
                balances[msg.sender][2] = 10000; /// a green coin
                balances[msg.sender][0] = 10000; /// a red coin
                balances[msg.sender][1] = 10000; /// a orange
coin
                balances[msg.sender][2] = 10000; /// a green coin
        }
        function sendCoinFunction(address coinsReceiver, uint
coinsAmount, uint coin) {
                if (balances[msg.sender][coin] < coinsAmount)
return;
                balances[msg.sender][coin] -= coinsAmount;
                balances[coinsReceiver][coin] += coinsAmount;
        }
}
```

If you copy the above contract into the Alethzero's new transaction window, you will see some additional 4-byte ABI code which has been generated by the compiler. This will be called "balances". A transaction could execute this function, but, this would never be. The accessor functions are normally accessed via message calls from the other contracts and calls are made from the web front-end.

Chapter 4

Getting Information from Blockchain

We will be exploring how the JavaScript API can be used to build string front ends for Dapps. We will also discuss how the JavaScript API can be used to pull information from both the client and blockchain stored on the local computer.

The following JavaScript template will be used for this task, and new functions will be added later to it. No custom libraries will be used in this case, but we can use bootstrap to have things look a bit nicer. Here is the template:

```
<html>

<head>

  <title>JavaScript API</title>

  <link rel="stylesheet"
href="http://maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstra

  <link rel="stylesheet"
href="http://maxcdn.bootstrapcdn.com/bootstrap/3.2.0/js/bootstrap

<script type="text/javascript">

///1st web3.eth.watch code for monitoring coinbase



///Add your contract address variable here
```

```
///2nd web3.eth.watch code for monitoring block number


</script>

</head>

<body>

  <div class="header">

    <h3>JavaScript</h3>

  </div>

  <div class="jumbotron">

    <h5>The Coinbase Address: <strong id="coinbase"></strong>
</h5>

    <h5>Available Balance: <strong id="coinBalance"></strong>
</h5>

    <h5>Latest Block Number: <strong id="blockNumber"></strong>
</h5>

    <h5>The Latest Block Timestamp: <strong id="timeStamp">
</strong></h5>

    <h5>The Latest Block Hash: <strong id="blockHash"></strong>
</h5>

    <h5>The Contract String: <strong id="contString"></strong>
</h5>

    <h5>The Favorite Python: <strong id="favPython"></strong>
</h5>

    <br>

  </div>

</body>

</html>
```

Just copy the code and paste it into a text editor of choice. Save the file as an HTML file. Launch Alethzero then use the browser to navigate to the file.

The fields shown in the template are blank, but we will use JavaScript bindings to pull information from the browser then pass it to the web page.

All our bindings are using the web3 object, and we only use the web3.eth object. We will first use the web3.eth.watch binding. This is an object which monitors the state of the blockchain and helps one to specify what triggers the function by use of filters. We have two options for this, including “chain” and “pending”, and they look out for the changes in the blockchain or for any pending transactions respectively. Log items can also be specified as the filters.

Paste the code given below into HTML, save the page and reload the AZ web browser:

```
web3.eth.watch('chain').changed(function(){

  //Add your code here

});
```

The watch object function will execute once changes are made to the blockchain, that is after a new block has been mined. Currently, there is no code for execution, but we can modify it to the following:

```
web3.eth.watch('chain').changed(function(){

  var coinbase = web3.eth.coinbase;

  document.getElementById('coinbase').innerText = coinbase;

});
```

We have created the “coinbase” variable which will help us to retrieve the coinbase account from the client by use of web3.eth.coinbase. The coinbase account is the one we will use to mine the blocks and send the transaction. Once the state of the blockchain changes, the web page will be modified by the function to display the coinbase account number. However, our coinbase account will not change that much. It will only change after adding a new account or after manually setting the coinbase to something different. The function “web3.eth.balanceA” will return the balance in ether for a particular address in a string format. This will then be used to show the string balance at

the coinbase address:

```
web3.eth.watch('chain').changed(function(){

  var coinbase = web3.eth.coinbase;

  document.getElementById('coinbase').innerText = coinbase;

  var coinBalance = web3.eth.balanceAt(coinbase);

  document.getElementById('coinBalance').innerText = coinBalance;

});
```

Just open the web page in a browser. For those with something higher than a zero balance, you will see something such as “0x678”. This is because the account balance is returned in the form of a hexadecimal value. The built-in function named “web3.toDecimal()” is responsible for doing the conversion to a hexadecimal value:

```
var coinBalance = web3.eth.balanceAt(coinbase);

document.getElementById('coinBalance').innerText =
web3.toDecimal(coinBalance);
```

Refresh the web page on your browser and you will see values in the fields for “Coinbase Address” and “Balance”.

Since both functions have been placed inside the “web3.eth.watch” function, after registering a new block, the code will be executed again, and the web page will be updated. You can mine some more ether and watch this as it updates.

The “watch” can be used for monitoring the blockchain as well as various filters which can be used to change what triggers the execution of the code. By use of the “pending” filter, the watch object will register the change both after mining a new block and after a new transaction has entered the pending queue.

We will use another watch object with “pending” filter to monitor the blockchain and update the web page with information on the latest block. This can be done by calling “web3.eth.number” and this will give us the

number of blocks which have been mined of late. This can then be passed as an argument to the “web3.eth.block” method and it will, in turn, give us a JSON object with numerous parameters having all relevant information regarding the block. We will take two of highly useful “timestamp” and “hash” and this will return the time when the latest block was mined as well as the hash of the block:

```
web3.eth.watch('pending').changed(function(){

  var blockNumber = web3.eth.number;

  document.getElementById('blockNumber').innerText = blockNumber;

  var hash = web3.eth.block(blockNumber).hash;

  document.getElementById('blockHash').innerText = hash;

  var timeStamp = web3.eth.block(blockNumber).timestamp;

  document.getElementById('timeStamp').innerText =
Date(timeStamp);

});
```

Six lines of code will be executed once the watch object has registered a change, the first and second will take the block number and insert it into the new webpage. The four final ones will pull the parameters from JSON object “result” then modify the webpage using them. Refresh the webpage then mine some more blocks and you will be able to see the values being updated in a real time manner.

Lastly, we can use the API for a more useful purpose, that is, pulling information from the contracts data storage. We should first create a contract then initialize it with data in its storage:

```
contract OurContract {

  string32 firstName;

  string32 lastName;

  string32 twitter;

  string32 nick;
```

```
  address address1;
  uint joinDate;
  function OurContract(){
    firstName = 'Joel';
    lastName = 'Boss';
    twitter = '@Boss';
    nick ='BlueChain';
    address1 = msg.sender;
    joinDate = block.timestamp;
  }
}
```

Copy the code given above into the window for “New Transaction” then send it to blockchain in a transaction in the same way you did previously. While it is in the pending pane, just get the contracts address since you will have to add it to JavaScript.

```
var contAddress  = "0x62ae0988e11437678c3a75f89e487efffb101a7e";
```

This will be used together with “web3.eth.storageAt()” function to retrieve the contracts data storage. Instead of creating another watch object, we can make an addition to our second “watch” which is monitoring “pending” queue. However, we should know that the contracts storage cannot change without us creating a new block:

```
web3.eth.watch('pending').changed(function(){
  var blockNumber = web3.eth.number;
  document.getElementById('blockNumber').innerText = blockNumber;
  var hash = web3.eth.block(blockNumber).hash;
  document.getElementById('blockHash').innerText = hash;
  var timeStamp = web3.eth.block(blockNumber).timestamp;
```

```
  document.getElementById('timestamp').innerText =
Date(timeStamp);

  var contString = JSON.stringify(web3.eth.storageAt(contAddress
))

  document.getElementById('contString').innerText = contString;

});
```

The “web3.eth.storageAt” function will return a JSON object which will have all the contents of our contracts storage. The JSON object in this case is simply a list of key values which may be different for each contract.

The strings send to the contract will be returned in the form of hex code, so we can get a set of key-value pairs then convert it to ASCII. This can be done using the “web3.toAscii” function as demonstrated below:

```
...

  var storageObject = web3.eth.storageAt(contAddress);

  document.getElementById('fullName').innerText =
web3.toAscii(storageObject['0x']) + ' ' +
web3.toAscii(storageObject['0x01']);

  ...
```

This has completed our webpage.

Chapter 5

Sending Transactions

It is good for you to note that contracts are equal in privileges and abilities to users outside the network. Anything that any user with a valid private key is capable of doing, a contract can also do. This includes the creation of new contracts, sending and receiving information from the other contracts and sending the transactions with ether.

Sending Transactions

In this section, we will be creating a simple Dapp that will raise a particular coinsAmount of ether after a fixed period. This will help in demonstrating how contracts handle ether, and how they normally send transactions.

Data Structuring

Before we can write a contract, you should first consider the kind of information that you will store in the contracts storage as well as the structure of the data. In this case, we will be demonstrating a crowdfunding campaign. We will be using a contract in which the contributions are made in transactions to a contract with ether, with the goal of the fundraising being in the ether, and blocks will be used to measure the allotted time. The contract should be able to handle multiple crowdfunded campaigns at once, keep track of every contribution made, evaluate whether the campaign was successful and pay the proposer or refund the investor after the expiry of the time limit.

Let us first consider the information which is related to the campaign we are doing:

- CampaignStruct ID
- Recipient of funds
- FundingStruct Goal
- Time Limit

With the above information, we can define the terms of our campaign. Information about the campaigns website or goals could be added feasibly but we will leave it to the front-end. For each campaign, there will be investors, so we should store the following:

- Investor account
- Amount invested

We should use a 2 * n mapping to store the above, where n will be the number of investments which have been made in the campaign as well as the total contributions. With this, we will be able to refund the investors in case a campaign fails to reach its goal.

```
We can then go ahead and come up with pairs of structs:

struct FundingStruct {

    address add;

    uint coinsAmount;

}

struct CampaignStruct {

    address beneficiaryAddress;

    uint  fundingGoal;

    uint numbeOfFunders;
```

```
    uint coinsAmount;

    uint deadlineForCampaign;

    mapping (uint => FundingStruct) funders;

}
```

We have defined our first struct with the name “FundingStruct” and it has two members namely “add” and “coinsAmount”. These two will represent the address of the contributor as well as the size of their contribution. The second struct is the “CampaignStruct” and this has five members. These will store the data related to the campaign in a storage, while the sixth element will be a mapping with previously defined struct datatype, that is, FundingStruct.

Lastly, we have declared a mapping of datatype CampaignStruct and this will be named “campaigns” and a stored integer named numOfCampaigns. The value of this will increment by one each time a new campaign is created:

```
mapping (uint => CampaignStruct) campaigns;

uint numOfCampaigns;
```

At this point, we have a structured way which can be used for storage of data during the period of the campaign.

Contracts Functions

We should now define the functions that our contract will need to perform and the changes that will be caused to the state of contract transactions calling the functions. In our previous example, we only used a single function to send coins from one account to another. In this case, we will be defining several transactions for this.

From the definition of a crowdsource, it can be broken down into three types

of operation which are triggered by three different types of transactions:

Proposer's transaction:

This is the transaction that will be sent by the proposer who creates the campaign. It includes the goal coinsAmount, recipient account and campaign time limit. These will all be immutable until the time that the time limit expires.

Investors transactions:

Investors sent by the investor can be ether and campaign ID for the campaign being funded. The transaction will be recorded as an investment in storage space and the sent ether will be added to total contribution.

End Of CampaignStruct transaction:

After expiry of the time, any transaction sent to the contract should trigger the end of fundraising. A transaction should be sent to trigger the action as contracts do not run continuously on a blockchain and they must be "woken" in order to run. The transaction will lead to the contract assessing whether the goal of the fundraising has been reached or not, after which it will choose to deliver the proceeds to proposer or refund the coinsAmount which was sent by the investor.

The first function intends to create a new campaign:

```
function aNewCampaign(address beneficiaryAddress, uint goal, uint
deadlineForCampaign) returns (uint campID) {

    campID = numOfCampaigns++;

    CampaignStruct camp = campaigns[campID];

    camp.beneficiaryAddress = beneficiaryAddress;

    camp. fundingGoal = goal;

    camp.deadlineForCampaign = block.number +
deadlineForCampaign;
```

```
}
```

Our function takes three arguments, and it will return a single value, that is, campID which has been defined within the function. The three arguments to the function aNewCampaign() include beneficiaryAddress, goal, and deadlineForCampaign. The numOfCampaigns is also incremented by 1. This will mark the first change to contracts storage.

We have also created the reference "camp" pointing to a location in contracts storage. This has then been used to assign parameters for the campaign in the storage. The "block.number" will give us the value of the current contract which the code is executing within. The deadline for the campaign has been set as "deadlineForCampaign" blocks in the future.

The execution of the function will set our contract ready for contributions from the investors. The second function will allow the investors to add contributions simply by sending the transactions to the contract:

```
function  contributeFunction(uint campID) {

    CampaignStruct camp = campaigns[campID];

    FundingStruct f = camp.amfunders[camp.numbeOfFunders++];

    f.add = msg.sender;

    f.coinsAmount = msg.value;

    camp.coinsAmount += f.coinsAmount;

}
```

The function is taking only one argument, that is, the campID, but each function is also passed some two additional arguments, that is, msg.sender and msg.value. To explain what is happening in our code:

In the first line, we have created a function which takes only a single argument campID. It has no return value.

In lines 2 and 3, we have created references to the locations in storage to store a new funder and iterate the number of the contributors. In lines 4, 5 and 6, we have taken the coinsAmount of ether which has been passed with

contributors' transaction and public address of msg.sender then use it for recording the contribution from the transaction. This is of importance because in any case, the campaign fails to meet the target, then the ether can be returned to the sender.

A function is needed for ending the event, and we will be creating it in our next step. There are two conditions for ending a campaign. First, the time limit may run out. Secondly, the goal of the fundraising may be reached. The final function should be large, but it is good for you to recognize how it works:

```
function goalCheckFunction(uint campID) returns (bool reached) {

    CampaignStruct camp = campaigns[campID];

    if (camp.coinsAmount >= camp. fundingGoal){

        camp.beneficiaryAddress.send(camp.coinsAmount);

        camp.coinsAmount = 0;

        camp.beneficiaryAddress = 0;

        camp. fundingGoal = 0;

        camp.deadlineForCampaign = 0;

        uint i = 0;

        uint f = camp.numbeOfFunders;

        camp.numbeOfFunders = 0;

        while (i <= f){

            camp.funders[i].add = 0;

            camp.funders[i].coinsAmount = 0;

            i++;

        }

        return true;

    }
```

```
    if (camp.deadlineForCampaign <= block.number){
        uint j = 0;
        uint n = c.numbeOfFunders;
        camp.beneficiaryAddress = 0;
        camp. fundingGoal = 0;
        camp.numbeOfFunders = 0;
        camp.deadlineForCampaign = 0;
        camp.coinsAmount = 0;
        while (j <= n){
            camp.funders[j].add.send(c.funders[j].coinsAmount);
            camp.funders[j].add = 0;
            camp.funders[j].coinsAmount = 0;
            j++;
        }
        return true;
    }
    return false;
}
```

We have two conditionals which will run sequentially. The first if will check whether the fundraising goal has been reached. If this has been reached, then the total coinsAmount raised will be send by the function to the address of the beneficiary which was defined in the first function. Consider the line given below:

camp.beneficiaryAddress.send(camp.coinsAmount);

The line above will also zero out the storage space which the campaign was using since it is not needed anymore. Finally, a true will be returned and the execution of the code will be terminated.

The second if will only be executed if our first conditional evaluated to a false and the fundraising goal has not been reached. It will have to check whether the deadline which has been defined in the setup of the campaign has passed. The following line is responsible for checking this:

```
if (camp.deadlineForCampaign <= block.number)
```

If it has passed, then it will loop through the contributor's list and return their ether to them. This will also zero out the storage space used by the campaign since it is no longer needed. Lastly, a true will be returned and the execution of the code will be terminated.

If no condition is met, the function will return a false and tell the caller the campaign has not met any of the conditions required to end it. The complete contract should be as shown below:

```
contract CrowdFundingContract {

    // data structure for holding information about the campaign
contributors

    struct FundingStruct {

        address add;

        uint coinsAmount;

    }

    // The campaign data structure

    struct CampaignStruct {

        address beneficiaryAddress;

        uint  fundingGoal;

        uint numbeOfFunders;

        uint coinsAmount;

        uint deadlineForCampaign;

        mapping (uint => FundingStruct) funders;

    }
```

```
    //Declaring the state variable 'numOfCampaigns'

    uint numOfCampaigns;

    //Creating a mapping for CampaignStruct datatypes

    mapping (uint => CampaignStruct) campaigns;

    //first function sets up a new campaign

    function aNewCampaign(address beneficiaryAddress, uint goal,
uint deadlineForCampaign) returns (uint campID) {

        campID = numOfCampaigns++; // campID is the return
variable

        CampaignStruct camp = campaigns[campID]; // will assign a
reference

        camp.beneficiaryAddress = beneficiaryAddress;

        camp. fundingGoal = goal;

        camp.deadlineForCampaign = block.number +
deadlineForCampaign;

    }

    //function contributing to the campaign

    function  contributeFunction(uint campID) {

        CampaignStruct camp = campaigns[campID];

        FundingStruct f = camp.funders[camp.numbeOfFunders++];

        f.add = msg.sender;

        f.coinsAmount = msg.value;

        camp.coinsAmount += f.coinsAmount;

    }

// checks whether the goal/time limit has been reached then end
the campaign

    function goalCheckFunction(uint campID) returns (bool
reached) {
```

```
CampaignStruct camp = campaigns[campID];
if (camp.coinsAmount >= camp. fundingGoal){
    uint i = 0;
    uint f = camp.numbeOfFunders;
    camp.beneficiaryAddress.send(camp.coinsAmount);
    camp.coinsAmount = 0;
    camp.beneficiaryAddress = 0;
    camp. fundingGoal = 0;
    camp.deadlineForCampaign = 0;
    camp.numbeOfFunders = 0;
    while (i <= f){
        camp.funders[i].add = 0;
        camp.funders[i].coinsAmount = 0;
        i++;
    }
return true;
}
if (camp.deadlineForCampaign <= block.number){
    uint j = 0;
    uint n = camp.numbeOfFunders;
    camp.beneficiaryAddress = 0;
    camp. fundingGoal = 0;
    camp.numbeOfFunders = 0;
    camp.deadlineForCampaign = 0;
    camp.coinsAmount = 0;
```

```
            while (j <= n){

camp.funders[j].add.send(camp.funders[j].coinsAmount);
                camp.funders[j].add = 0;
                camp.funders[j].coinsAmount = 0;
                j++;
            }
            return true;
        }
        return false;
    }
}
```

The contract should be uploaded to the blockchain in Alethzero. Ensure that you have some ether then open the "New Transaction" window. Copy then paste the contracts code into "Data" box and you will see the contracts ABI in compile box beneath. Copy then paste the function ID into the text document for a later use then click "Execute".

Mine until the pending transaction disappears and you will see the contract in the contract's pane. You can now try the contract by creating new accounts for yourself or by using a dummy address for the recipient. Use the aNewCampaign function and add a new campaign.

You should then contribute to the campaign yourself which will be different. You will then have a choice to either fully fund your campaign before you can trigger the end of the campaign by use of "goalCheckFunction" function or force mine some few more blocks until the deadline has passed and the campaign fails, and the funds have been returned to the contributors. Regardless of the option you choose, the campaign should disappear from the contracts storage once you are done.

Chapter 6

Amending Blockchain State

In this chapter, we will be discussing how one can amend the state of the blockchains. The only way for us to interact with blockchains is by sending transactions. These are of three main types including transactions for sending value, transactions for creating contracts and transactions for sending data to those transactions.

You can use the HTML template given below but fill your own JavaScript code to the template:

```
<!DOCTYPE html>

<html lang="en">

        <head>

                <meta charset="UTF-8">

                <title>JSAPI.html</title>

                <a href="home.html">Home Page</a>

                <script type="text/javascript">


                //contract source code


                //contract abi object


                //function for simple transaction to transfer
ether
```

```
                //function to create new contract from source
code


                //function to send transaction by use of
attributes from HTML input boxes

                </script>

        </head>

        <body>

        <div>

        <div>

         <h3>Transact</h3>

             </div>

                <div>

                <input id="recipient" type="text"
placeholder="To">

                <input id="value" type="text" placeholder="Value
in Wei">

                </div>

                <div>

                <button onclick="sendEtherFunction();">Send
Ether</button>

                </div>

                <div>

                <h3>Create a contract</h3>

                </div>

                <div>

                <button onclick="newSolidityContract();">Create
```

```
Solidity Coin Contract</button>

                </div>

                <div>

        <h3>Send coins</h3>

        </div>

        <div>

        <input id="receiverAddress" class="form-control"
type="text" placeholder="Receiver address"></input><br>

        <input id="coinsAmount" class="form-control" type="text"
placeholder="Amount"></input><br>

        </div>

        <button onclick="dataTransFunction();">Send
transaction</button>

        </div>

        <div>

        Add addresses prefixed with '0x' then hit 'Send
transaction'

        </div>

        </body>

</html>
```

Just copy the above code and paste it into HTML file, save the file in your workspace then open it in Alethzero.

We will be using the “web3.eth.transact()” JavaScript function which takes a JSON object as the argument and this JSON argument has numerous possible attributes. We should only be concerned with two attributes which are “to” and “value” and these are recipient address and coinsAmount of ether that we need to send, and the calculation of this is done in the base unit named “Wei”. Transact is simply an asynchronous function whose callback returns various pieces of information.

Copy the following into your HTML template then open the webpage in the Alethzero:

```
function sendEtherFunction() {

web3.eth.transact({to: document.querySelector('#recipient').value
,value: document.querySelector('#value').value});

};
```

A JSON object with the attributes “to” and “value’ is passed to the transact function. The values for these two attributes are pulled from the form input. It is possible for us to include a gas and gasPrice for the attributes, but these will default to the total coinsAmount of ether that is available and the current mean price on our network. These are enough for our example.

Copy then paste an address into the box for “To” and any coinsAmount of wei that you need to send in the field for coinsAmount. Click transact. When you hit the button, you will call the “sendEtherFunction” function which will, in turn, instruct the client to send the transaction to the network.

You will get a popup warning you that the webpage is requesting the client to send a transaction. Hit the approve button and if everything is okay, you will see the transaction appear on the pending pane. Ensure that you mine a block to commit it to the chain and check to see whether the balance update is correct.

Our next transaction will be passing the EVM code to the blockchain with no specified recipient to create a new contract. Before we can do this, we should have an EVM code which will be passed to the transaction. The JavaScript API can compile the solidity into byte code so that it can be passed into an array. We will first create a JavaScript object with code for Solidity metaCoinContract. Add the following to your HTML template:

```
var source = "contract metaCoinContract {\n"+

" mapping (address => uint) balances;\n" +

" function metaCoinContract() {\n" +

" balances[msg.sender] = 10000;\n" +
```

```
" }\n" +

" function sendCoinFunction(address coinsReceiver, uint
coinsAmount) returns(bool successful) {\n" +

" if (balances[msg.sender] < coinsAmount) return false;\n" +

" balances[msg.sender] -= coinsAmount;\n" +

" balances[coinsReceiver] += coinsAmount;\n" +

" return true;\n" +

" }\n" +

"}";
```

We used this code previously and uploaded it to the blockchain by use of the transact pane for Alethzero. This time, the code will be compiled in the browser and uploaded as a bytecode. The function to be compiled from solidity will be "web3.eth.solidity()" and once it is passed, the solidity code will produce the EVM bytecode. Just do the following:

Copy then paste the above code in your html template. Reload the page on your Alethzero then run the following command on JavaScript console:

```
web3.eth.solidity(source)
```

We will then get a bytecode which can be passed as a parameter to the transact function by use of "code:" This is shown below:

```
web3.eth.transact({code:
'0x70016011563b60ae8060326000396000f35b61271060003600160a060010a
```

The contract will be uploaded to the blockchain in the same way as putting solidity in transact pane. It will create a callback which will in return give the address of the created contract.

Paste the following code into your HTML document:

```
function newSolidityContract() {

web3.eth.transact({code: web3.eth.solidity(source)});
```

```
};
```

This will allow us to create a contract with a single click from the browser window. The method can be used for compiling and deploying any contract needed through the browser. The “web3.eth.serpent()” method can also be used for compiling serpent code in a similar way.

Try it by opening HTML page in Alethzero then click “create solidity contract” button. You will see a popup warning you that you are sending a transaction with zero wei to the network. Just approve this and you will see it in the pending pane.

You should now be aware of how to interact with contracts by use of transact pane in Alethzero by building a data array for passing the contracts. The JavaScript API will abstract this process and allow you to directly call the contracts.

First, we will model the contracts functions by definition of an array of JSON objects which corresponds to the functions that our contract has. Each object will represent a function of the contract and it is made up of specific attribute-value pairs. The first attribute will be “name” and its case sensitive to values and it must take the functionName(argumentType1, argumentType2,…) format. For our above contract, this should look as follows:

```
"name": "sendCoinFunction(address,uint256)",

The second argument denotes the type, and in our case, the type
will be “function”:

"type": "function",

Lastly, we should define the inputs for the function an array of
objects, where both will the attributes “name” and “type”:

"inputs": [

{

"name": "coinsReceiver",

"type": "address"

},
```

```
{
"name": "coinsAmount",
"type": "uint256"
}
]
```

In case the function returned a value, it would have been possible for us to define “outputs” so that the JavaScript can know how to handle data array which is passed back to our browser by a callback.

When we put it all together, we create a JavaScript object which defines our contract as follows:

```
var contractDescription = [{
"name": "sendCoinFunction(address,uint256)",
"type": "function",
"inputs": [
{
"name": "to",
"type": "address"
},
{
"name": "value",
"type": "uint256"
}
]
}];
```

Note the curly braces which have been used to indicate the objects, while the square brackets have been used to mark arrays. At this point, we have a

description of the contracts function and the arguments it takes. We can use the “web3.eth.contract()” function to create a contract object taking two arguments. The first argument will be the address of the contract which will take a 160-bit hex string, while the second one will be a contract description:

```
contract = web3.eth.contract(address, contractDescription);
```

We can then change our “newSolidityContract()” function to create the object:

```
var contractDescription = [{

"name": "sendCoinFunction(address,uint256)",

"type": "function",

"inputs": [

{

"name": "to",

"type": "address"

},

{

"name": "value",

"type": "uint256"

}

]

}];



...



function newSolidityContract() {

var address = web3.eth.transact({code:
web3.eth.solidity(source)});
```

```
contract = web3.eth.contract(address, contractDescription);

};
```

Several important changes have been made. First, a new variable named "address" has been declared, and this will be the value which is returned by the transact function that creates our contract. Secondly, a new global object named "contract" has been declared, and this will be created after passing the contract description object together with the contracts address to the "web3.eth.contract" function.

It is now easy for us to interact with the contract on the blockchain using the new contract object by calling the functions we have defined above:

```
contract.sendCoinFunction('0x678932387afe76fd008e7665','500')
```

The above sends a transaction instructing our contract to send 500 coins to the count "0x678932387afe76fd008e7665". It is easy for us to create a JavaScript function that will take these arguments from the HTML form and pass them to the contract as follows:

```
function dataTransFunction() {

var receiverAddress =
document.querySelector('#receiverAddress').value;

var coinsAmount = document.querySelector('#coinsAmount').value;

contract.send(receiverAddress, coinsAmount);

};
```

When the above code is executed, it will generate a pop-up window with a warning that the client has received a request from the front end to send the transaction that includes data. The warning will also indicate that the client is not aware of the effect the data may have on the state of the contracts and the effect can be more profound instead of just a transfer of ether or gas costs.

After mining of a new block, just check the state of the contract. The changes to the storage should be registered inside it. That's it!

Chapter 7

Function Visibility

Contracts consist of state variables and functions, but so far, we have been executing functions by use of message calls from outside the contract. However, it is possible for us to have our own contracts to call their own functions directly which can use this to simplify our code.

Let us go back to our crowdfund contract:

```
contract CrowdFundingContract {

    // data structure for holding information about the campaign
contributors

    struct FundingStruct {

        address add;

        uint coinsAmount;

    }

    // The campaign data structure

    struct CampaignStruct {

        address beneficiaryAddress;

        uint  fundingGoal;

        uint numbeOfFunders;

        uint coinsAmount;

        uint deadlineForCampaign;

        mapping (uint => FundingStruct) funders;
```

```
    }

    //Declaring the state variable 'numOfCampaigns'

    uint numOfCampaigns;

    //Creating a mapping for CampaignStruct datatypes

    mapping (uint => CampaignStruct) campaigns;

    //first function sets up a new campaign

    function aNewCampaign(address beneficiaryAddress, uint goal,
uint deadlineForCampaign) returns (uint campID) {

        campID = numOfCampaigns++; // campID is the return
variable

        CampaignStruct camp = campaigns[campID]; // will assign a
reference

        camp.beneficiaryAddress = beneficiaryAddress;

        camp. fundingGoal = goal;

        camp.deadlineForCampaign = block.number +
deadlineForCampaign;

    }

    //function contributing to the campaign

    function  contributeFunction(uint campID) {

        CampaignStruct camp = campaigns[campID];

        FundingStruct f = camp.funders[camp.numbeOfFunders++];

        f.add = msg.sender;

        f.coinsAmount = msg.value;

        camp.coinsAmount += f.coinsAmount;

    }

// checks whether the goal/time limit has been reached then end
the campaign

    function goalCheckFunction(uint campID) returns (bool
```

```
reached) {
        CampaignStruct camp = campaigns[campID];
        if (camp.coinsAmount >= camp. fundingGoal){
            uint i = 0;
            uint f = camp.numbeOfFunders;
            camp.beneficiaryAddress.send(camp.coinsAmount);
            camp.coinsAmount = 0;
            camp.beneficiaryAddress = 0;
            camp. fundingGoal = 0;
            camp.deadlineForCampaign = 0;
            camp.numbeOfFunders = 0;
            while (i <= f){
                camp.funders[i].add = 0;
                camp.funders[i].coinsAmount = 0;
                i++;
            }
        return true;
        }
        if (camp.deadlineForCampaign <= block.number){
            uint j = 0;
            uint n = camp.numbeOfFunders;
            camp.beneficiaryAddress = 0;
            camp. fundingGoal = 0;
            camp.numbeOfFunders = 0;
            camp.deadlineForCampaign = 0;
```

```
            camp.coinsAmount = 0;
            while (j <= n){
camp.funders[j].add.send(camp.funders[j].coinsAmount);
                camp.funders[j].add = 0;
                camp.funders[j].coinsAmount = 0;
                j++;
            }
            return true;
        }
        return false;
    }
}
```

In the above example, we zero out the entries for each campaign at the end of each event. The code can be simplified by creating a function which will perform this for us:

```
contract CrowdFundingContract {
    // data structure for holding information about the campaign
contributors
    struct FundingStruct {
        address add;
        uint coinsAmount;
    }
    // The campaign data structure
    struct CampaignStruct {
        address beneficiaryAddress;
        uint  fundingGoal;
```

```
        uint numbeOfFunders;
        uint coinsAmount;
        uint deadlineForCampaign;
        mapping (uint => FundingStruct) funders;
    }
    //Define the state variable 'numOfCampaigns'
    uint numOfCampaigns;
    //Create the mapping of the CampaignStruct datatypes
    mapping (uint => CampaignStruct) campaigns;
    //first function sets up a new campaign
    function aNewCampaign(address beneficiaryAddress, uint goal,
uint deadlineForCampaign) returns (uint campID) {
        campID = numOfCampaigns++;
        CampaignStruct camp = campaigns[campID];
        camp.beneficiaryAddress = beneficiaryAddress;
        camp. fundingGoal = goal;
        camp.deadlineForCampaign = block.number +
deadlineForCampaign;
    }
    function  contributeFunction(uint campID) {
        CampaignStruct camp = campaigns[campID];
        FundingStruct f = camp.funders[camp.numbeOfFunders++];
        f.add = msg.sender;
        f.coinsAmount = msg.value;
        camp.coinsAmount += f.coinsAmount;
    }
```

```
    function goalCheckFunction(uint campID) returns (bool
reached) {
        CampaignStruct camp = campaigns[campID];
        if (camp.coinsAmount >= camp. fundingGoal){
            camp.beneficiaryAddress.send(camp.coinsAmount);
            cleanFunction(campID);
                return true;
        }
        if (camp.deadlineForCampaign <= block.number){
            uint j = 0;
            uint n = camp.numbeOfFunders;
            while (j <= n){

camp.funders[j].add.send(camp.funders[j].coinsAmount);
                j++;
            }
            cleanFunction(campID)
            return true;
        }
        return false;
    }
    function cleanFunction(uint id) {
        CampaignStruct camp = campaigns[id];
        uint i = 0;
        uint n = camp.numfunders;
        camp.coinsAmount = 0;
```

```
        camp.beneficiaryAddress = 0;
        camp. fundingGoal = 0;
        camp.deadlineForCampaign = 0;
        camp.numbeOfFunders = 0;
        while (i <= n){
            camp.funders[i].add = 0;
            camp.funders[i].coinsAmount = 0;
            i++;
        }
    }
}
```

The contract now works but it has a problem. During the compilation of contracts, Solidity assumes that you need all the functions to be accessible from outside via ABI. However, we don't need our outside actors to be able to "clean" our campaigns storage at will. This means that the function should be marked for internal use only:

```
function cleanFunction(uint id) private {
        CampaignStruct camp = campaigns[id];
        uint i = 0;
        uint n = camp.numfunders;
        camp.coinsAmount = 0;
        camp.beneficiaryAddress = 0;
        camp. fundingGoal = 0;
        camp.deadlineForCampaign = 0;
        camp.numbeOfFunders = 0;
        while (i <= n){
```

```
            camp.funders[i].add = 0;
            camp.funders[i].coinsAmount = 0;
            i++;
        }
```

The cleanFunction() function is now a private function, meaning that only the contract itself can access it. The function cannot be called externally, and after adding it to the Alethzero, you will see that no ABI function ID will be created.

To specify the visibility of a function, one can use any of the four options including external, inherited, public and private. The default one is public, and you don't have to specify it. It will allow for the function to be accessed internally or via ABI. When a function is specified as private, only the contract itself will be able to access the function. When specified as inherited, the function will be accessible by the contract as well as the derivatives of the contract. When set as external, the function will only be accessible via message calls.

If you fail to specify the visibility of a function correctly, the contract may not operate as specified. You should consider when your functions amend the state of the contract.

You are now aware of how to create accessor functions to the state variables simply by specifying it as public. A storage can have three possible visibility statuses including private, inherited and public. The default one is inherited, and it allows access to a storage by the contract and its derivatives.

Global variables

The home for contracts is the blockchain. To send information from the outside world to the contracts, you can use transactions and they will interact with it as their specification demands. However, the information will only be as trustworthy as the source which provided it. This way, one can be in a

situation where the usefulness of the contract is limited to there being a very little trusted information for it to run the logic on.

If your contract is just a simple metacoin, this will matter very little. The only information that the contract logic needs to assess is kept in coin balances in storage. If you are having enough coins, you can choose to transfer some. However, this is not the case with many Dapps.

The consensus mechanism employed in the Ethereum blockchain provides contracts with information regarding the state of the blockchain which can be trusted. Below is a list of the global variables:

- block.coinbase (address)- current address of block miner
- block.gaslimit (uint)- the current block gaslimit
- block.number (uint)- the current block number
- block.difficulty (uint)- the current block difficulty
- block.blockhash (function(uint) returns (hash))- the hash of a given block
- msg.data (bytes)- the complete calldata
- msg.gas (uint)- the remaining gas
- block.timestamp (uint)- the current block timestamp
- msg.sender (address)- the sender of the message (current call)
- tx.gasprice (uint)- the gas price of a transaction
- msg.value (uint)- the number of wei sent with a message
- tx.origin (address)- the sender of transaction (full call chain)
- add.balance (uint)- where the address of an account in the Ethereum network

- this.balance (uint- the current contract balance

The variables given above will return information to the contract about the state of the blockchain, and the contracts executing them can consider them as a “truth”.

Chapter 8

Contracts Interactions

Contracts should be viewed as first class members of the Ethereum network with similar privileges as external actors. They have two important abilities including the ability to issue message calls to the other contracts and the ability to issue new contracts themselves.

In this chapter, we will be building two contracts, the coinCallerContract which will interact with metaCoinContract contracts and mint responsible for issuing contracts. Our metaCoinContract contract is as follows:

```
contract metaCoinContract {

        mapping (address => uint) balances;

        function metaCoinContract() {

                balances[msg.sender] = 10000;

        }

        function sendCoinFunction(address coinsReceiver, uint
coinsAmount) returns(bool enough) {

                if (balances[msg.sender] < coinsAmount) return
false;

                balances[msg.sender] -= coinsAmount;

                balances[coinsReceiver] += coinsAmount;

                return true;

        }

}
```

This is just a simple contract with a single constructor function, and one function which others can call.

Calling another Contract

Whenever a contract talks to another contract, it makes use of same ABI that we must use to call a function. For us to build a contract capable of calling another contract, the compiler should know the source code of the second contract so that our contract may know how to access its functions.

For us to do this, we should first define the contract that we need to call then follow it with the contract we need to create. This is demonstrated below:

```
contract metaCoinContract {

        mapping (address => uint) public balances;

        function metaCoinContract() {

                balances[msg.sender] = 10000;

        }

        function sendToken(address coinsReceiver, uint
coinsAmount) returns(bool successful){

                if (balances[msg.sender] < coinsAmount) return
false;

                balances[msg.sender] -= coinsAmount;

                balances[coinsReceiver] += coinsAmount;

                return false;

        }

}


contract coinCallerContract{
```

```
        //code to be added here

}
```

You can paste the above code in Alethzero and you will notice that the compiled code that you get will be for a contract named coinCallerContract and it has no functions. When a series of functions are passed, the compiler will assume that the contract you need to deploy to blockchain is the final one, and any other contracts which are received will be ignored until they are referenced in the main contract. That is what we need to do now.

We will add some code to our coinCallerContract function which will call the sendToken function inside the metaCoinContract function given an address:

```
contract metaCoinContract {

        ...

        }

}

contract coinCallerContract{

        function sendCoinFunction(address coinContractAddress,
address coinsReceiver, uint coinsAmount){

                metaCoinContract met =
metaCoinContract(coinContractAddress);

                met.sendToken(coinsReceiver, coinsAmount);

        }

}
```

We have the sendCoinFunction function which takes three arguments, that is, the metaCoinContract address, the recipient address and the number of coins you need to send. We have defined “met” a metaCoinContract contract by explicitly converting the address given to a metaCoinContract. We can now call the function “met”s functions by use of dot operator, then pass to it the arguments recipient and the coinsAmount.

After calling another contracts function, in the same manner, you will be

having the contract send similar data array which we used earlier. If the address passed is not a valid metaCoinContract contract, the message will still be sent, but the code will not be executed.

You can deploy the contract in Alethzero together with a metaCoinContract contract. You will have to first give the address of the coinCallerContract some coins since it will not have any coins. If this is done correctly, you should be able to see the metaCoinContract contract in the same way that it could if the functions were called directly.

Like the other datatypes, the contracts can be held in a storage then used as datatypes in the structs and mappings. Example:

```
contract metaCoinContract {

        ...

        }

}

contract coinCallerContract{

        struct transferStruct{

                metaCoinContract coinContract;

                uint coinsAmount;

                address recipient;

        }

        mapping(uint => transferStruct) transfers;

        uint numOfTransfers;

        function sendCoinFunction(address coinContractAddress,
address coinsReceiver, uint coinsAmount){

transferStruct trans = transfers[numOfTransfers]; //to create
reference trans

                trans.coinContract =
metaCoinContract(coinContractAddress);
```

```
                trans.coinsAmount = coinsAmount;

                trans.recipient = coinsReceiver;

                trans.coinContract.sendToken(coinsReceiver,
coinsAmount);

                numOfTransfers++;

        }

}
```

In the above example, each of the transfers will be stored in coinCallerContract's storage including metaCoinContract contract which we sent the transfer to. Again, deploy it with metaCoinContract contract and you will see the transfer being written to coinCallerContracts storage.

At this point, you should be aware of how to declare contract types then send message calls, but message calls are designed to return data to a sender. Our metaCoinContract contract is having two functions for returning data. Our sendCoinFunction function will be returning a Boolean based on whether the balance is sufficient to send coins. The accessor function "balance" will return the balance of coins. We can now rewrite our contract so that it can handle data which is returned from the metaCoinContract:

```
contract metaCoinContract {

        ...

        }

}


contract coinCallerContract{

        struct transferStruct{

                metaCoinContract coinContract;

                uint coinsAmount;

                address recipient;
```

```
                bool successful;

                uint coinBalance;

        }

        mapping(uint => transferStruct) transfers;

        uint numOfTransfers;

        function sendCoinFunction(address coinContractAddress,
address coinsReceiver, uint coinsAmount){

transferStruct trans = transfers[numOfTransfers]; //Create a
reference trans trans.coinContract =
metaCoinContract(coinContractAddress);

        trans.coinsAmount = coinsAmount;

        trans.recipient = coinsReceiver;

        trans.successful =
trans.coinContract.sendToken(coinsReceiver, coinsAmount);

        trans.coinBalance = trans.coinContract.balances(this);

        numOfTransfers++;

        }

}
```

After sending a message call to a metaCoinContract contract, the return value of the sendCoinFunction function is stored and this is simply a Boolean value which states that the transfer was successful. Another message call has been added to the accessor function of metaCoinContract's "balances" as shown below:

```
trans.coinBalance = trans.coinContract.balances(this);

This is stored alongside the other data on our transaction.
```

Creating a New Contract

Contracts, just like external factors, can create new contracts. Previously, we discussed how the Solidity compiler needs access to the source code of a contract whose functions we intend to call. This also applies if we need to give a contract the ability to create some new contract. The contract should know the whole of the source code and the whole of it will be uploaded to the blockchain after deployment.

Let us begin by creating a new contract named “coinSpawnContracter” that will produce metaCoinContract contracts. Here is the contract:

```
contract metaCoinContract {

        mapping (address => uint) public balances;

        function metaCoinContract() {

                balances[tx.origin] = 10000;

        }

        function sendToken(address coinsReceiver, uint
coinsAmount) returns(bool successful){

                if (balances[msg.sender] < coinsAmount) return
false;

                balances[msg.sender] -= coinsAmount;

                balances[coinsReceiver] += coinsAmount;

                return false;

        }

}


contract coinSpawnContract{

        function createCoinFunction(){

                new metaCoinContract();

        }

}
```

The “createCoinFunction” function in the coinSpawnContract will create a metaCoinContract contract when it is called by use of a function ID. Note that we can call the function several times.

Note that an alteration has been made to the metaCoinContract contract in line 4, that is:

```
balances[tx.origin] = 10000;
```

The “msg.sender” has been replaced with “tx.origin” for the contract to credit the sender of original transaction but not the sender of message call.

Once a contract has been created, the address of the contract is returned, to the creator contract. We can set up the coinSpawnContract contract to handle this. This is shown below:

```
contract metaCoinContract {

        mapping (address => uint) public balances;

        function metaCoinContract() {

                balances[tx.origin] = 10000;

        }

        function sendToken(address coinsReceiver, uint
coinsAmount) returns(bool successful){

                if (balances[msg.sender] < coinsAmount) return
false;

                balances[msg.sender] -= coinsAmount;

                balances[coinsReceiver] += coinsAmount;

                return false;

        }

}

contract coinSpawnContract{

        mapping(uint => metaCoinContract)
readilyDeployedContracts;
```

```
        uint numContracts;

        function createCoinFunction() returns(address add){

                readilyDeployedContracts[numContracts] = new
metaCoinContract();

                numContracts++;

                return readilyDeployedContracts[numContracts];

        }

}
```

Although the source code of the contract that is to be created should be set at compile time, the coinSpawnContract contract can pass the constructor function arguments as long as you have set it up to receive them:

```
contract metaCoinContract {

        mapping (address => uint) public balances;

        function metaCoinContract(uint initialCoinBalance) {

                balances[tx.origin] = initialCoinBalance;

        }

        function sendToken(address coinsReceiver, uint
coinsAmount) returns(bool successful){

                if (balances[msg.sender] < coinsAmount) return
false;

                balances[msg.sender] -= coinsAmount;

                balances[coinsReceiver] += coinsAmount;

                return false;

        }

 }


contract coinSpawnContract{
```

```
        mapping(uint => metaCoinContract)
readilyDeployedContracts;

        uint numContracts;

        function createCoinFunction(uint initialCoinBalance)
returns(address add){

                readilyDeployedContracts[numContracts] = new
metaCoinContract(initialCoinBalance);

                numContracts++;

                return readilyDeployedContracts[numContracts];

        }

}
```

At this point, the coinSpawnContract contract can deploy the metaCoinContract contracts for users at will, and the users should specify their starting balance. You can deploy the above in Alethzero then send it a transaction to deploy a metaCoinContract.

Chapter 9

Two-Party Contracts

Contracts can be seen as trustless decentralized escrow, where contracts enable transfers between parties by acting as third parties.

In this chapter, we will discuss how hashes are calculated in contracts and some simple verification activities outside the blockchain.

We will use a simple scenario involving our two actors, Bob and Alice. Bob wants to buy a Piano, and Alice has a Piano for sale. Lice will begin posting the Piano on eBay, then Bob will see it from there. Bob will bid for the Piano, and if his bid wins, he will send the payment together with his delivery address. Alice will then deliver the Piano:

```
contract decentralisedAuctionContract{

        struct auction {

                uint deadlineForCampaign;

                uint highBid;

                address highBidder;

                address recipient;

        }

        mapping(uint => auction) Auctions;

        uint numberOfAuctions;


        function launchAuction(uint timeLimit) returns (uint
auctionID){
```

```
                auctionID = numberOfAuctions++;

                Auctions[auctionID].deadlineForCampaign =
block.number + timeLimit;

                Auctions[auctionID].recipient = msg.sender;

        }

        function bid(uint id) returns (address highBidder){

                auction au = Auctions[id];

                if (au.highBid + 1*10^18 > msg.value ||
au.deadlineForCampaign > block.number) {

                        msg.sender.send(msg.value);

                        return au.highBidder;

                }

                au.highBidder.send(au.highBid);

                au.highBidder = msg.sender;

                au.highBid = msg.value;

                return msg.sender;

        }

        function  terminateAuction(uint id) returns (address
highBidder){

                auction au = Auctions[id];

                if (block.number >= au.deadlineForCampaign) {

                        au.recipient.send(au.highBid);

                        au.highBid = 0;

                        au.highBidder =0;

                        au.deadlineForCampaign = 0;

                        au.recipient = 0;

                }
```

```
        }
}
```

This will be familiar to you because we have written the crowdFund contract previously, which was to handle ether payments and deal with timelimits for action purposes.

Bidders will have to send their maximum contributions they need to bid to the contract and any time that the limit is up, the seller will be able to collect their winning. However, in this case, we are not protecting Bob to ensure Alice delivers the Piano. For us to do this, we should use the contract as an escrow to hold ether as a third party and wait for either of the following two things to happen:

- The timelimit runs out.
- Alice delivers the Piano.

The contract will be implemented in such a way that Bob will have to send a hash value when bidding, and only him should know this hash. After the end of the auction, the contract will hold the ether until it has received a transaction from someone, most probably Alice and her proxy. A correct random number will be sent to it, or a pre-image, and ether will be sent to Alice. If the contract fails to receive the pre-image within the right time, the contract will be voided, and the ether will be returned to Bob.

In a real-world scenario, Alice may hire a courier firm who will carry the Piano and deliver it to Bob. In this case, the courier firm should be instructed to only give the Piano to Bob once he provides the pre-image that meets the SHA256(key) = HASH. After getting the key, they may choose to send the transaction to blockchain themselves and the will be released.

The new contract is shown below:

```
contract decentralisedAuctionContract{

        struct auction {

                uint deadlineForCampaign;
```

```
                uint highBid;

                address highBidder;

                uint bidHash;

                address recipient;

        }

        mapping(uint => auction) Auctions;

        uint numberOfAuctions;


        function launchAuction(uint timeLimit) returns (uint
auctionID){

                auctionID = numberOfAuctions++;

                Auctions[auctionID].deadlineForCampaign =
block.number + timeLimit;

                Auctions[auctionID].recipient = msg.sender;

        }

        function bid(uint id, uint biddersHash) returns (address
highBidder){

                auction au = Auctions[id];

                if (au.highBid + 1*10^18 > msg.value ||
au.deadlineForCampaign > block.number) {

                        msg.sender.send(msg.value);

                        return au.highBidder;

                }

                au.highBidder.send(au.highBid);

                au.highBidder = msg.sender;

                au.highBid = msg.value;

                au.bidHash  = biddersHash;
```

```
                return msg.sender;

        }

        function  terminateAuction(uint id, uint key) returns
(address highBidder){

                auction au = Auctions[id];

                if (block.number >= au.deadlineForCampaign &&
sha3(key) == au.bidHash) {

                        au.recipient.send(au.highBid);

                        cleanFunction(id)

                }

        }

        function cleanFunction(uint id) private{

                auction au = Auctions[id];

                au.highBid = 0;

                au.highBidder =0;

                au.deadlineForCampaign = 0;

                au.recipient = 0;

                au.bidHash = 0;

        }

}
```

We are waiting for an event to occur for the secret key to be revealed. A similar contract can be implemented whereby we will wait for a message from Bob to release the funds from escrow to Alice, but we will have to wait for Bob to send the transaction.

Alice will courier the Piano to the eventual winner so adds some third party to the contract:

```
contract decentralisedAuctionContract{
```

```
        struct auction {
                uint deadlineForCampaign;
                uint highBid;
                address highBidder;
                uint bidHash;
                address recipient;
                address thirdPartyAddress;
                uint thirdPartyCharge;
        }
        mapping(uint => auction) auctions;
        uint numberOfAuctions;

        function launchAuction(uint timeLimit, address
thirdPartyAddress, uint thirdPartyCharge) returns (uint
auctionID){
                uint ID = numberOfAuctions++;
                auction au = auctions[ID];
                au.deadlineForCampaign = block.number +
timeLimit;
                au.recipient = msg.sender;
                au.thirdPartyAddress = thirdPartyAddress;
                au.thirdPartyCharge = thirdPartyCharge;
        }
        function bid(uint id, uint biddersHash) returns (address
highBidder){
                auction au = auctions[id];
                if (au.highBid + 1*10^18 > msg.value ||
au.deadlineForCampaign > block.number) {
```

```
                        msg.sender.send(msg.value);

                        return au.highBidder;

                }

                au.highBidder.send(au.highBid);

                au.highBidder = msg.sender;

                au.highBid = msg.value;

                au.bidHash  = biddersHash;

                return msg.sender;

        }

        function  terminateAuction(uint id, uint key) returns
(address highBidder){

                auction au = auctions[id];

                if (block.number >= au.deadlineForCampaign &&
sha3(key) == au.bidHash) {

                        au.recipient.send(au.highBid-
au.thirdPartyCharge);

                        au.thirdPartyAddress.send(au.thirdPartyCh

                        cleanFunction(id)

                }

        }

        function cleanFunction(uint id) private{

                auction au = Auctions[id];

                au.highBid = 0;

                au.highBidder =0;

                au.deadlineForCampaign = 0;

                au.recipient = 0;

                au.bidHash = 0;
```

```
                au.thirdPartyAddress = 0;

                au.thirdPartyCharge = 0;

        }

}
```

The third party is simply a courier who simply receives fee from the contract in the same way as Alice, by simply getting the secret pre-image from Bob. The delivery will only be completed once Bob has handed over the right secret pre-image.

However, our contract faces a challenge. What will happen in case Alice decides to send the Piano to Bob? Bob's money will be trapped forever. We should create an event with a time limit so that in case Bob's secret is not revealed within the time limit, he will be able to reclaim his ether:

```
contract decentralisedAuctionContract{

        struct auction {

                uint deadlineForCampaign;

                uint highBid;

                address highBidder;

                uint bidHash;

                address recipient;

                address thirdPartyAddress;

                uint thirdPartyCharge;

                uint deadlineForDelivery;

        }

        mapping(uint => auction) auctions;

        uint numberOfAuctions;


        function launchAuction(uint timeLimit, address
```

```
thirdPartyAddress, uint thirdPartyCharge, uint
deadlineForDelivery) returns (uint auctionID){

                uint ID = numberOfAuctions++;

                auction au = auctions[ID];

                au.deadlineForCampaign = block.number +
timeLimit;

                au.recipient = msg.sender;

                au.thirdPartyAddress = thirdPartyAddress;

                au.thirdPartyCharge = thirdPartyCharge;

                au.deadlineForDelivery = block.number + timeLimit
+ deadlineForDelivery;

        }

        function bid(uint id, uint biddersHash) returns (address
highBidder){

                auction au = auctions[id];

                if (au.highBid + 1*10^18 > msg.value ||
au.deadlineForCampaign > block.number) {

                        msg.sender.send(msg.value);

                        return au.highBidder;

                }

                au.highBidder.send(au.highBid);

                au.highBidder = msg.sender;

                au.highBid = msg.value;

                au.bidHash  = biddersHash;

                return msg.sender;

        }

        function  terminateAuction(uint id, uint key) returns
(address highBidder){
```

```
                auction au = auctions[id];

                if (block.number >= au.deadlineForCampaign &&
sha3(key) == au.bidHash) {

                        au.recipient.send(au.highBid-
au.thirdPartyCharge);

                        au.thirdPartyAddress.send(au.thirdPartyCh

                        cleanFunction(id);

                }

        }

        function failedDelivery(uint id) {

                auction au = auctions[id];

                if (block.number >= au.deadlineForDelivery &&
msg.sender == au.highBidder){

                        au.highBidder.send(au.highBid);

                        cleanFunction(id);

                }

        }

        function cleanFunction(uint id) private{

                auction au = Auctions[id];

                au.highBid = 0;

                au.highBidder =0;

                au.deadlineForCampaign = 0;

                au.deadlineForDelivery = 0;

                au.recipient = 0;

                au.bidHash = 0;

                au.thirdPartyCharge = 0;

                au.thirdPartyAddress = 0;
```

```
        }
}
```

At this point, we have a contract that has been setup to handle the real-world transactions over time.

Where else can Secrets be used?

Consider the contract given below:

```
contract transfer{
        uint hashOfOwner;
        address ownerAddress;
        function transfer(){
                hashOfOwner =
0xea1d340a2b5a99864b3946af168cd361760a5e413f4939c9f5d3cbd3c6e10cd
                ownerAddress = msg.sender;
        }
        function receiveCoinFunction(uint key){
                if (sha3(key) == hashOfOwner){
                        ownerAddress.send(this.coinBalance);
                }
        }
}
```

The contract is a constructor function storing a hash passed during the deployment and together with sender's address. It has only one function, that is, "receiveCoinFunction" that will send the balance of the contract to the contracts owner after the correct pre-image has been given.

The contract has been designed to be deployed by two different users. Their goal is for the two users to be able to exchange ether. The deployment steps

should be as follows:

1. Alice will deploy a copy of the contract to our blockchain together with zero ether. Only Alice is aware of the “preimage” of hashOfOwner:

```
contract transfer{

        uint hashOfOwner;

        address ownerAddress;


        function transfer(){

                hashOfOwner =
0xea1d340a2b5a99864b3946af168cd361760a5e413f4939c9f5d3cbd3c6e10cd

                ownerAddress = msg.sender;

        }

        function receiveCoinFunction(uint key){

                if (sha3(key) == hashOfOwner){

                        ownerAddress.send(this.coinBalance);

                }

        }

}
```

2. Bob will see the contract on Blockchain then send an exact copy of the contract. This will have the same hash that was used in the first contract whose pre-image is only known by Alice.

3. Alice sends 500 ether to the contract for Bob then she waits. Bob will see this then send 500 ether to the contract for Alice.

4. Since Alice possesses the secret key, she can send a transaction with the key to some newly funded contract, and the contract will send her the ether.

5. Once the transaction is sent, Alice will have revealed the pre-image that

Bob can use in his contract to have it send him ether.

Note that we have used a convoluted method to transfer ether between our two parties when it is very easy for us to use a single contract. This is a good technique because there is no need for the two contracts to be running on a similar blockchain. This method can also be modified easily to allow for a trustless exchange of the other cryptocurrencies between the chains. The methods become much useful during the development of Dapps that involve multiple parties since publishing pre-images of a hash is simpler compared to signing transactions using private keys.

Chapter 10

Events & Logging in Solidity

Logs and events are essential in Ethereum since they facilitate communication between smart contracts and their corresponding user interfaces. In the traditional web development, the server response is provided in a callback to the frontend. In Ethereum, after mining of a transaction, the smart contract can emit events then write logs to the blockchain that the frontend will be able to process. The events and logs can be addressed in various ways.

Events may confuse you since there are different ways through which they can be used. One event may be different from another event. The following are the three use cases of events:

1. Asynchronous triggers with data.

2. Smart contract return values for user interface.

3. A cheaper storage medium.

Smart Contract Return Values for User Interface

The simplest use of an event is passing along return values from contracts to the frontend of an app. Consider the following contract:

```
contract ContractExample {

  //state variables ...
```

```
  function func(int256 _value) returns (int256) {
    // manipulate the state ...
    return _value;
  }
}
```

Assuming we have “contractExample” as an instance of ContractExample, a frontend that uses web3.js can get a return value by simulating the execution of the contract:

var valueOfReturn = contractExample.func.call(2);

console.log(valueOfReturn) // 2

However, in case the web3.js submits the contract call in the form of a transaction, it can’t get the return value. This is demonstrated below:

```
var valueOfReturn = contractExample.func.sendTransaction(2,
{from: web3.eth.coinbase});

console.log(valueOfReturn) //the transaction hash

The return value for the sendTransaction method should be the
hash of the created transaction. Transactions do not return a
contract value to frontend since transactions are not mined
immediately and included in the blockchain.

The best solution is to use an event which is one of the reasons
for having events:

contract ContractExample {

  event ValueOfReturn(address indexed _from, int256 _value);

function func(int256 _value) returns (int256) {

    ValueOfReturn(msg.sender, _value);

    return _value;

  }

}
```

We can then use the frontend in order to get the return value as follows:

```
var eventExample = contractExample.ValueOfReturn({_from:
web3.eth.coinbase});

eventExample.watch(function(anError, result) {

  if (anError) {

    console.log(anError)

    return;

  }

  console.log(result.args._value)

  // check whether result.args._from is web3.eth.coinbase then

  // show result.args._value on the UI and call

  // eventExample.stopWatching()

})

contractExample.func.sendTransaction(2, {from:
web3.eth.coinbase})
```

After mining the transaction calling func, we will trigger the callback inside the watch. This will allow the frontend to get return values from the func.

Asynchronous Triggers with Data.

Return values are one of the use cases for events, and events may be considered to be asynchronous triggers with data. When a contract needs to trigger the frontend, the contract will emit an event. Since the frontend is watching for events, it can display a message, take an action etc.

A cheaper Storage Medium

Events can be used as a cheaper form of storage. Events can be seen as logs. When talking about storage, we can say that data can be stored in logs instead of events. Contracts normally emit or trigger events and the frontend can react to this. After emission of an event, the corresponding logs are written to the blockchain.

Logs were designed to act as a cheaper form of storage compared to the contract storage in terms of usage of gas. The logs normally cost 8 gas for every byte, while a contract storage will cost 20,000 gas for every 32 bytes. However, although provide a cheaper way of storage, they cannot be accessed from any contract.

A cryptocurrency exchange may need to show the user all the deposits that may have been made on the exchange. Instead of having to store the details of the deposit in the contract, it is a bit cheaper to store these in logs. This is made possible by the fact that an exchange needs to know the state of the user's balance, which is stored in the contract storage. However, it does not need to know details about the historical deposits. Example:

```
contract CryptocurrencyExchangeContract {

  event EventDeposit(uint256 indexed _market, address indexed
_sender, uint256 _coinsAmount, uint256 _time);

function depositFunc(uint256 _coinsAmount, uint256 _market)
returns (int256) {

    // perform the deposit, then update the user's balance, etc

    EventDeposit(_market, msg.sender, _coinsAmount, now);

}
```

Suppose we need to update the UI as the user makes the deposits. The following example shows the use of the event (EventDeposit) as an asynchronous trigger with the data (_market, msg.sender, _coinsAmount, now). Assume that the cryptocurrencyExamContract is an instance of CryptocurrencyExchangeContract:

```
var depositEvent =
cryptocurrencyExamContract.EventDeposit({_sender:
addressOfUser});
```

```
depositEvent.watch(function(anError, result) {
  if (anError) {
    console.log(anError)
    return;
  }
  // append the details of result.args to the UI
})
```

The _sender parameter to the event has been indexed to improve the efficiency of getting all the events of a user.

By default, listening to an event is only triggered after the instantiation of an event. At the time the UI is loading for the first time, no deposits to append to it. So, we need to retrieve events since the block 0, and this can be done by adding the “fromBlock” parameter to our event. This is shown below:

```
var depositAll =
cryptocurrencyExamContract.EventDeposit({_sender: addressOfUser},
{fromBlock: 0, toBlock: 'latest'});
 depositAll .watch(function(anError, result) {
  if (anError) {
    console.log(anError)
    return;
  }
  // append the details of result.args to the UI
})
```

Once the UI has been rendered, the method “depositAll .stopWatching()” will be called. Note that we can index up to 3 parameters.

As discussed in the chapter, events can be used to get a return value from the contract function invoked with the sendTransaction() function. Secondly, an

event can be used as an asynchronous trigger with some data, capable of notifying an observer like a UI. Thirdly, events can be used to write logs in a blockchain as a cheaper form of storage.

Logging Access Using Events

Security is implemented by use of Authentication, Authorization, and Auditing mechanisms. Authentication is provided by Ethereum once a user has initiated a transaction with a particular account. The assumption is that the user owns the account as he can use it to initiate a transaction. The authentication of the user is done at the Ethereum node client level.

For the case of authorization, role-based access control is used. For the case of auditing, it is required that the user should be logged.

Just like normal instance variables of a contract, events can be inherited in the same way as any variable. Events are used when there is a need to return values to the client, but for now, we can use them as a means of adding logs to the blockchain. When events are fired, they form part of the blockchain together with the parameters supplied to them.

Let us define an event variable that will help us implement a log event:

event LogAccessEvent(address indexed by, uint indexed accessTime, string method, string description);

The event will be used in access control method isUserFunc() to log access that is being attempted with its result:

```
function isUserFunc(address candidateAddress, string method)
returns (bool){

        for(uint8 x = 0; x < users.length; x++){

            if (users[x] == candidateAddress){

              LogAccessEvent(candidateAddress, now, method,
"successful access");
```

```
                return true;

            }

        }

      LogAccessEvent(candidateAddress, now,method,  "failed
access");

      return false;

    }
```

The event has parameterized the accessing account (candidate), the resource (method), time (now) and the result (failed access, successful access). A string has been used to describe the result, but it is always good for one to use uint constants to make the search much easier.

The event will be fired every time the isUserFunc() is fired and the access will be logged to the blockchain to help in the auditing purposes. A web3 client can be used to access the events.

Chapter 11

Inheritance

Solidity supports multiple inheritance simply by copying the code including polymorphism. All function calls in Solidity are virtual, meaning that the most derived function is called, except in cases where the name of the contract is given explicitly.

In case a contract inherits from many other contracts, only one contract will be created in the blockchain, and the code from base contracts will be copied into the contract that has been created.

Consider the following contract:

```
contract contOwned {

    function contOwned () { ownerAddress = msg.sender; }

    address ownerAddress;

}

// Use "is" to derive from some other contract. Derived

// contracts are able to access all non-private members including

// the internal functions and the state variables. They cannot be

// accessed externally using `this`.

contract mortal is contOwned {

    function kill() {

        if (msg.sender == ownerAddrress)
selfdestruct(ownerAddress);
```

```
    }
}
// These abstract contracts are provided to make the
//compiler know the interface. Note the function
// with no body. If a contract doesn't implement all
//the functions, we can only use it as an interface.
contract Config {
    function lookupFunc(uint id) returns (address adr);
}

contract NameReg {
    function register(bytes32 name);
    function unregister();
 }

// Multiple inheritance is supported. "owned" is
// also a base class of the "mortal", yet we only have a single
// instance of "owned".

contract named is owned, mortal {
    function named(bytes32 name) {
        Config config =
Config(0xc4f9d8d35886e70b05e474c3fb14fd43e2f23963);
        NameReg(config.lookupFunc(1)).register(name);
    }
```

```
    // A function can be overridden by another function having
the same //name and same number/types of parameters.  If the
overriding function //has different types of output parameters,
it will cause an error.

    // Both local and the message-based function calls will take
these //overrides into account.

    function kill() {

        if (msg.sender == ownerAddress) {

            Config config =
Config(0xc4f9d8d35886e70b05e474c3fb14fd43e2f23963);

            NameReg(config.lookupFunc(1)).unregister();

            // It is possible to call a certain

            // overridden function.

            mortal.kill();

        }

    }

}


// In case a constructor takes an argument, it should be

// provided in the header

contract PriceFeed is owned, mortal, named("GoldFeed") {

   function updateInfo(uint newInfo) {

      if (msg.sender == ownerAddress) info = newInfo;

   }


   function get() constant returns(uint r) { return info; }


   uint info;
```

```
}
```

In the above example, we are calling “mortal.kill()” in order to “forward” the destruction request. This is done in a problematic way as shown below:

```
contract owned {

    function owned() { ownerAddress = msg.sender; }

    address ownerAddress;

}


contract mortal is owned {

    function kill() {

        if (msg.sender == ownerAddress)
selfdestruct(ownerAddress);

    }

}


contract Base1 is mortal {

    function kill() { /*perform cleanup 1 */ mortal.kill(); }

}




contract Base2 is mortal {

    function kill() { /* perform cleanup 2 */ mortal.kill(); }

}



contract Final is Base1, Base2 {
```

```
}
```

When the “Final.kill()” is called, it should, in turn, call “Base2.kill” since it’s the most derived override, but the function will bypass “Base1.kill” simply it doesn’t know about “Base1”. We can use “super” in order to solve this:

```
contract owned {

    function owned() { ownerAddress = msg.sender; }

    address ownerAddress;

}


contract mortal is owned {

    function kill() {

        if (msg.sender == ownerAddress)
selfdestruct(ownerAddress);

    }

}

contract Base1 is mortal {

    function kill() { /* perform cleanup 1 */ super.kill(); }

}


contract Base2 is mortal {

    function kill() { /* perform cleanup 2 */ super.kill(); }

}


contract Final is Base2, Base1 {

}
```

If the “Base1” calls a function of “super”, it doesn’t simply call this function

on any of its base contracts. However, it calls the function on the next base contract in final inheritance graph, meaning that it will call “Base2.kill()”. When using super, the real function that is called is not known in the context of the class in which it has been used, although the type of the function is known. This is the same with ordinary virtual method lookupFunc.

Base Constructor’s Arguments

Derived contracts should provide all arguments that are needed for the base constructors. There are two ways in which this can be done:

```
contract Base {
    uint x;
    function Base(uint _x) { x = _x; }
}

contract Derived is Base(7) {
    function Derived(uint _y) Base(_y * _y) {
    }
}
```

One way is by doing it directly in the inheritance list (is Base(7)). The other method is in the way a modifier is invoked as part of the header of the derived constructor (Base(_y * _y)). The first way of doing this becomes more convenient where the constructor argument is a constant and it defines the contract’s behavior, or it describes it. The second method should be used when the constructor arguments of the base is dependent on those of derived contract. If both places are used, then the modifier-style argument will take precedence.

Multiple Inheritance and Linearization

Languages capable of supporting multiple inheritance should have means to deal with many problems. One of these is the Diamond problem. Solidity follows Python path and it uses “C3 Linearization” in order to force a particular order in DAG of base classes. This leads to monotonicity, which is a desirable property, but it does not allow for some inheritance graphs. The order the base classes are given to the “is” directive is very important. Consider the following code that is expected to give an error:

```
contract X {}

contract A is X {}

contract C is A, X {}
```

The code will not compile, but it will instead give the error “Linearization of inheritance graph impossible”. This is caused by the fact that “c” will request “x” to override “A”, but “A” itself is requesting to override “x”. This brings about a contradiction that cannot be solved. Remember the rule of specifying the base classes in the order from the “most base-like” to the “most derived”.

When inheritance gives a contract with a function and modifier of the same name, it is an error. The error is produced by an event and modifier of the same name, and a function and event of the same name. A state variable getter can override a public function.

Abstract Contracts

Contract functions may lack an implementation as demonstrated below:

```
pragma solidity ^0.4.0;

contract Feline {
```

```
    function utterance() returns (bytes32);
}
```

Note that we have used a semicolon to terminate the function declaration header. These types of contracts cannot be compiled but we can use them as base contracts. This applies even when they have implemented functions together with non-implemented functions:

```
pragma solidity ^0.4.0;


contract Feline {
    function utterance() returns (bytes32);
}


contract Cat is Feline {
    function utterance() returns (bytes32) { return "miaow"; }
}
```

If a contract inherits from an abstract contract and it doesn't implement all the non-implemented functions through overriding, it will become abstract itself.

Interfaces

Interfaces are the same as abstract methods, but they cannot have functions implemented. Other restrictions on interfaces include the following:

1. Cannot inherit the other interfaces or contracts.
2. Cannot define variables.
3. Cannot define a constructor.

4. Cannot define enums.

5. Cannot define structs.

However, some of them may be lifted in the future. Interfaces are limited to what contract ABI can represent, and conversations between interfaces and the ABI should be possible without a loss of information. To create interfaces, we use the “interface” keyword as demonstrated below:

```
pragma solidity ^0.4.11;


interface Token {

    function transfer(address recipient, uint coinsAmount);

}
```

Contracts are capable of inheriting interfaces in the same way they inherit other contracts.

Chapter 12

Libraries

Libraries are the same as contracts, but they are only deployed once at a specific address and their code is reused using the “DELEGATECALL” feature of Ethereum Virtual Machine (EVM). This means that after calling library functions, their code must be executed in the context of the contract that is calling, that is, “this” points to the contract that is calling, and the storage from calling contract may be accessed.

Since a library is an isolated piece of source code, it is only capable of accessing the state variables of the calling contract if they are supplied explicitly, otherwise, it will have no way of naming them.

Libraries may be seen as implicit base contracts of the contracts using them. They will not be explicitly visible in the in inheritance hierarchy but calls to the library functions look similarly to calls to functions of the explicit base contracts. Internal library functions are visible in all the contracts just like the library is a base contract. Calls made to internal functions u8se internal calling convention, meaning that all the internal types may be passed, and memory types will not be copied but will be passed by reference. For this to be realized in EVM, code for the internal library functions and all the functions called from therein will be pulled into calling contract, and a normal “JUMP” call should be used rather than DELEGATEALL.

Libraries can be used as demonstrated in the following contract:

```
pragma solidity ^0.4.11;

library Set {

  // A new struct datatype will be defined that will be used for
```

```
  // holding its data in calling contract.
  struct Data { mapping(uint => bool) flags; }
  // The first parameter is of "storage
  // reference" type, thus, only its storage address instead of
  // its contents is passed as part of call.  This forms a
  // special feature of the library functions.  It is idiomatic
  // for one to call first parameter 'self', in case the function
may
  // be seen as a method of the object.
  function insert(Data storage self, uint value)
      returns (bool)
  {
      if (self.flags[value])
          return false; // already there
      self.flags[value] = true;
      return true;
  }

  function remove(Data storage self, uint value)
      returns (bool)
  {
      if (!self.flags[value])
          return false; // not there
      self.flags[value] = false;
      return true;
  }
```

```
  function contains(Data storage self, uint value)
      returns (bool)
  {
      return self.flags[value];
  }
}


contract C {
    Set.Data knownValues;


    function register(uint value) {
        // The library functions may be called with no
        // specific instance of library, since the
        // "instance" will be current contract.
        require(Set.insert(knownValues, value));
    }
// In the contract, we are able to access the known values.flags directly
}
```

However, this is not the only way of using libraries since they can be used without defining the struct data types. Functions can also work without storage reference parameters, and they may also have multiple storage reference parameters in any location.

The calls to the Set.contains, Set.remove and Set.insert are compiled as calls to external library/contract. For those who use libraries, it is good for you to ensure an actual external function call is done. The “msg.value”,

“msg.sender”, and “this” will retain their value in the call.

The example given below demonstrates how internal functions and memory types can be used in libraries to implement custom types with no overhead of the external function calls:

```
pragma solidity ^0.4.0;


library BigInt {

    struct bigint {

        uint[] limbs;

    }


    function fromUint(uint x) internal returns (bigint r) {

        r.limbs = new uint[](1);

        r.limbs[0] = x;

    }


    function add(bigint _a, bigint _b) internal returns (bigint
r) {

        r.limbs = new uint[](max(_a.limbs.length,
_b.limbs.length));

        uint carry = 0;

        for (uint i = 0; i < r.limbs.length; ++i) {

            uint a = limb(_a, i);

            uint b = limb(_b, i);

            r.limbs[i] = a + b + carry;

            if (a + b < a || (a + b == uint(-1) && carry > 0))
```

```
                carry = 1;
            else
                carry = 0;
        }
        if (carry > 0) {
            // bad, we should add a limb
            uint[] memory newLimbs = new uint[](r.limbs.length +
1);
            for (i = 0; i < r.limbs.length; ++i)
                newLimbs[i] = r.limbs[i];
            newLimbs[i] = carry;
            r.limbs = newLimbs;
        }
    }

    function limb(bigint _a, uint _limb) internal returns (uint)
{
        return _limb < _a.limbs.length ? _a.limbs[_limb] : 0;
    }

    function max(uint a, uint b) private returns (uint) {
        return a > b ? a : b;
    }
}

contract C {
```

```
    using BigInt for BigInt.bigint;


    function f() {
        var x = BigInt.fromUint(7);
        var y = BigInt.fromUint(uint(-1));
        var z = x.add(y);
    }
}
```

Since the compiler is unable to know the location the library will be deployed, the addresses should be filled into the final bytecode using a linker. If the addresses are not passed as arguments to the compiler, the compiled hex code will have placeholders of the form __Set_____, and "set represents the library name. One can fill the addresses manually by replacing the all the 40 symbols by hex encoding of the address of library contract.

When compared to contracts, libraries have the following restrictions:

- No state variables
- Cannot receive Ether
- Cannot inherit or be inherited

Using For

We can use the "using A for B;" directive to attach library functions (from library A) to any type (B). The libraries will receive the objects they had called as their first parameter.

When we use "using A for *;", the functions from library A will be attached to any type.

In both cases, all the functions, including the ones where the type of the first parameter doesn't match the object type will be attached. The checking of the type is done during the function call, and the function overload resolution is done.

The directive "using A for B;" only remains active in the current scope, which is limited to the contract for now, but it will be lifted to global scope later, so that once a module is included, the data types and the library functions will be available without the need to add extra code.

Consider the following example:

```
pragma solidity ^0.4.11;


library Set {
  struct Data { mapping(uint => bool) flags; }


  function insert(Data storage self, uint value)
      returns (bool)
  {
      if (self.flags[value])
        return false; // already there
      self.flags[value] = true;
      return true;
  }


  function remove(Data storage self, uint value)
      returns (bool)
  {
      if (!self.flags[value])
```

```
            return false; // not there
        self.flags[value] = false;
        return true;
    }

    function contains(Data storage self, uint value)
        returns (bool)
    {
        return self.flags[value];
    }
}

contract C {
    using Set for Set.Data; // this is the critical change
    Set.Data knownValues;

    function register(uint value) {
        // Here, all the variables of the type Set.Data have
        // the corresponding member functions.
        // The following function call is identical to
        // Set.insert(knownValues, value)
        require(knownValues.insert(value));
    }
}
```

Elementary types can also be extended in the same way as demonstrated below:

```
pragma solidity ^0.4.0;

library Search {

    function indexOf(uint[] storage self, uint value) returns
(uint) {

        for (uint x = 0; x < self.length; x++)

            if (self[x] == value) return x;

        return uint(-1);

    }

}


contract C {

    using Search for uint[];

    uint[] data;


    function append(uint value) {

        data.push(value);

    }


    function replace(uint _old, uint _new) {

        // This will do the library function call

        uint index = data.indexOf(_old);

        if (index == uint(-1))

            data.push(_new);

        else
```

```
            data[index] = _new;
    }
}
```

All the library calls are actual EVM function calls. After passing a memory or value types, a copy will be done, even a copy of “self” variable.

Chapter 13

Access Control Lists

Access Control is a way of controlling and restricting access to a resource. Most access control models in Solidity are centered around the user or the "principal" who is accessing the resource. In some situations, the resource may define the access to itself. However, the two common models of access control are the role-based and attribute-based access control.

Suppose we have a contract named "MyContract" that provides methods which can be used to manage/create a list of users then provide methods that can be used any specific user against the list. The assumption is that each contract user is an "address" type. In solidity, the address is special kind of variable that represents the account or wallet address. It can be used as a user credential since a user is only able to access a contract by use of his account address. It can also be assumed that any call from the address if authentic since the call can only be generated if the user has access to the private key of the account, which forms the basic authenticity mechanism on the Ethereum network.

The template for the contract will have two instance variables:

```
address public ownerAddress;

address [] public users;
```

The ownerAddress address represents the creator of the contract. This will be instantiated during a call to the constructor. This is a fair assumption since the owner should be the only one deploying the contract unless there is a need to delegate the tasks of the owner to another address, for which provisions will be made within the contract.

This means if the contract is named MyContract, the code given below will instantiate the address of the owner, and the owner will be made to be one of the users:

```
function MyContract() {

    ownerAddress = msg.sender;

    users.push(ownerAddress);

}
```

After instantiating the contract, some operation methods will be needed to administer the available list of users. The msg is an object of special type in Solidity responsible for holding data about sender or caller transaction. In our case, the deployer account will be the caller of the transaction who deploys the contract, so we will have to assign the ownerAddress address to the address of this caller:

```
function addNewUser(address userAddress){

    if (msg.sender != ownerAddress) throw;

    users.push(userAddress);

}


function getUser(uint x) constant returns (address){

    return users[x];

}


function getTotalUsers() constant returns (uint){

    return users.length;

}

function deleteUser(uint x){

    if (msg.sender != ownerAddress) throw;
```

```
    delete users[x];
}
```

The clients will use these methods to manage the list of users. Only the owner will be able to administer the list with the check:

```
if (msg.sender != ownerAddress) throw;
```

The code matches the caller's address and causes the transaction to fail with a throw in case a mismatch occurs.

For us to vet a user against the access control list, it should be matched with the list of the available addresses. We will implement a simple method that will accept a user address then try to match it with the list:

```
function isUserFunc(address candidateAddress, string method)
returns (bool){

  for(uint8 x = 0; x < users.length; x++){

            if (users[x] == candidateAddress){

              return true;

            }

         }

      return false;

    }
```

The logic can be understood easily if the user is found in the list, we will exit the loop with a true. If we run out of the elements contained in the list, we will return a false.

The contract that was created can be inherited by any other contract to reuse the ACL lists that are provided by it. Our data contract inherited it by use of the following construct:

```
contract ContractForData is MyContract {

To use ACL facility, we will intercept the method calls in the
```

```
contract with isUserFunc() :

function customerUpdateFunction (uint index, string customerName)
{

   if (isUserFunc(msg.sender, "customerUpdateFunction")) {

      if (index > count) throw;

      customers[index]. customerName = customerName;

   }

   else throw;

}



function customerUpdateFunctionStatus(uint index, uint
customerStatus) {

   if (isUserFunc(msg.sender, "customerUpdateFunction")) {

       if (index > count) throw;

       customers[index]. customerStatus = customerStatus;

   }

   else throw;

}



function createNewCustomer(

uint id,

string customerName,

uint dateOfBirth,

uint social){

   if (isUserFunc(msg.sender, "createNewCustomer")) {

      customers[count] = Customer(id, customerName, dateOfBirth,
social, pending);
```

```
        count++;

    }

    else throw;

}
```

We are supporting the access control restrictions to the methods that update our data. If the customer data had been encrypted within the contract, a restriction would be implemented on its read capabilities.

The contract is unable to access the events it creates, so for the events to be accessed, our web3 client service will be needed to create a call:

Suppose the ABI has constructed ABI and the address is its location on the blockchain:

```
var ContractForData = web3.eth.contract(abi).at(address);

var logsForEvents = ContractForData.allEvents().get();

console.log(logsForEvents);
```

Note that we are only retrieving the events for auditing purposes only in this case.

Attribute-based Access Control

We can now implement the attribute-based access control. This is a type of access control in which the authorized users are identified using an attribute. This will be a variation of the isUserFunc() call:

```
function isUserFuncAuthorized(address candidateAddress, string
method) returns (bool){

        for(uint8 x = 0; x < users.length; x++){

            if (candidateAddress.coinBalance > 100){
```

```
                LogAccessEvent(msg.sender, now, method, "successful
access");
                return true;
            }
        }
      LogAccessEvent(msg.sender, now,method,  "failed access");
      return false;
    }
```

In this example, only users with ether greater than 100 will be authorized.

Chapter 14

Gas and Transaction Costs

When a gas limit is set, it protects buggy (or malicious) code from running until the funds are depleted. When the gas and gas price are multiplied, one gets the maximum Wei to be used in executing the transaction. The value you specify for gasPrice is used by miners to rank the transactions for inclusion in the blockchain. The price of a unit gas in Wei is used to price the VM operations.

The gas expenditure that is incurred on running a transaction is bought by the ether you have in the account at a price that is specified in the transaction with gasPrice. In case you don't have all the ether needed for you to cover all the gas requirements to complete execution of the code, the processing will abort all the intermediate state changes then roll back to a pre-transaction snapshot. The gas that has been used up to the point the execution has stopped was used after all, and the ether balance of the account is reduced. The adjustment of the parameters can be done on the gas and gasPrice object fields of the transaction. The field for "value" is used in the same way as in the ether transfer transactions between normal accounts. In other words, transfer of funds can be done between any two accounts, either normal or contract. In case the contract runs out of funds, you will see an error about insufficient funds.

If you need to test or play around with contracts, it is recommended that you set up a private node/network or use a test network. If you set up a private node, it will be good for you to isolate it from the rest of the nodes. Once you mine, ensure that the transaction is included in the next block. The following command can help you see all the pending transactions:

```
eth.getABlock("pending", true).transactions
```

Blocks can be retrieved using the number/height or hash as follows:

```
genesisBlock = eth.getABlock(0)

eth.getABlock(genesisBlock.hash).hash == genesisBlock.hash

true
```

You can use “eth.blockNumber” to get the height of the current blockchain as well as the “latest” magic parameter to access the newest block or current head:

```
blockHeight = eth.blockNumber()

eth.getABlock("latest").hash ==
eth.getABlock(eth.blockNumber).hash

true
```

For any contract to be used on Ethereum, it should first be deployed but the deployment process consumes a large coinsAmount of gas. Suppose we create a contract that will assign an ID to each IoT (Internet of Things) device. Here is the code for the contract:

```
contract AssignIDContract {

    bytes32 private id;

    string private name;

    function AssignIDContract(bytes32 _id, string _name) {

        id = _id;

        name = _name;

    }

    function getDeviceId() constant returns(bytes32) {

        return id;

    }

    function getDeviceName() constant returns(string) {
```

```
        return name;

    }

}
```

The contract is very simple as we have only implemented a getter and a setter. When we try to create an instance of this with the Browser-Solidity, we get a transaction cost of 219,000 GAS. The Ether and GASPrice prices normally fluctuate, but assuming 1 GAS is 0.00000002 Ether and 1 Ether is $10:

0.00000002Ether x 219,000GAS x $10 = $0.0438

It will cost around 0.04 cents. This should be clear after looking at the "data" section for the "Web3 deploy". It is good to include all the data compiled for all the sources in the transaction to deploy the contract:

```
var somethingContract =
web3.eth.contract([{"constant":true,...}]);

var something = somethingContract.new(

{

  from: web3.eth.accounts[0],


data:'0x6060604052346100005 75b6103498061001860003960 00f36060...',

  gas: '4700000'

}, function (e, contract){

  console.log(e, contract);

  if (typeof contract.address !== 'undefined') {

      console.log('Contract mined! address: ' + contract.address
+ ' transactionHash: ' + contract.transactionHash);

  }

})
```

However, there is another problem associated with this. If many contracts are

deployed over time, in the same way, there will be no guarantee that the contracts will all have the same logic.

Solution to Gas Consumption Issue

It is assumed that all the code is included in the contract when it is being deployed, if any contracts require more logic, then a huge coinsAmount of Gas will be consumed. This means that if the part for the field is made as a structure independent of the contract, the contract should be deployed just once.

An object is created in the form of a structure object within a contract that is being deployed, then it is referenced by mapping to the ID. Consider the source code below:

```
contract AssignIDContract {

    struct AssignIDContractStruct {

        bytes32 id;

        string name;

    }

    mapping (bytes32 => AssignIDContractStruct) devices;


    function create(bytes32 _id, string _name) {

        devices[_id] = AssignIDContractStruct(_id, _name);

    }


    function getDeviceId(bytes32 _id) constant returns(bytes32) {

        return devices[_id].id;

    }
```

```
    function getDeviceName(bytes32 _id) constant returns(string)
{
        return devices[_id].name;
    }
}
```

During deployment, the above will require 275,234 GAS, but the cost for the creation of each subsequent object has been reduced to 63,250 GAS, which is 22.98%.

We have managed to reduce the coinsAmount of GAS that is to be used, but the method we have used has one problem.

After the deployment of a contract, an address that uniquely identifies that contract on the Ethereum network is assigned. For the case of structure, every instance address is not available, and it is good to refer to the instance of the structure using a unique key, and in our case, this the id field.

If the keys are duplicated, we should prevent the occurrence of a duplicate registration by using throw to throw an error. The duplicate check can be done as follows:

```
contract AssignIDContract {
    struct AssignIDContractStruct {
        bytes32 id;
        string name;
    }
    mapping (bytes32 => AssignIDContractStruct) devices;

    function create(bytes32 _id, string _name) {
        if (devices[_id].id != "") throw;
```

```
        devices[_id] = AssignIDContractStruct(_id, _name);
    }
    function getDeviceId(bytes32 _id) constant returns(bytes32) {
        return devices[_id].id;
    }

    function getDeviceName(bytes32 _id) constant returns(string)
{
        return devices[_id].name;
    }
}
```

In Ethereum, the processing of smart contract logic is done serially, so after creating objects in the way shown above, the duplicates can be eliminated.

Since the relation between logic and data part during deployment is the reason for the issue, it is good for us to represent the data part as another contract. However, this will result in the following:

```
contract AssignIDFactory {
    mapping (bytes32 => AssignIDContract) devices;

    function create(bytes32 _id, string _name) {
        devices[_id] = new AssignIDContract(_id, _name);
    }

    function get(bytes32 _id) constant returns(address) {
        return address(devices[_id]);
    }
```

```
}

contract AssignIDContract {

    bytes32 private id;

    string private name;

    function AssignIDContract(bytes32 _id, string _name) {

        id = _id;

        name = _name;

    }

    function getDeviceId() constant returns(bytes32) {

        return id;

    }

    function getDeviceName() constant returns(string) {

        return name;

    }

}
```

The generation and execution of SomeTHingFactory will consume 200,247 GAS. From this, the solution can be seen to not have contributed to a reduction in consumption of the GAS. Even if you use another contract to generate another contract, the same coinsAmount of GAS as the standard deployment pattern will be consumed.

Chapter 15

Version Upgrade

One of the characteristics of Ethereum is that once contracts have been deployed, they cannot be tampered with. This characteristic makes Smart Contracts suitable for use as contract logic. If it were not for this, there could be no difference between Smart Contracts and the traditional database.

However, you may realize that a Smart Contract has a bug in it. In such a case, the smart contract application should be modified while adhering the following conditions:

1. The deployed code should not be modified.

2. The existing data should continue to be available.

3. After fixing the bug, the smart contract should continue with the bug resolved.

4. Only the individual who provided the contract is allowed to fix the bug.

Using CNS

CNS stands for ContractNameService. It works in the same way as DNS (Domain Name Service). The contract name is mapped to the contract address and returns the address when given the name of the contract.

When we need to call a contract, we use the address that corresponds to the name of the contract in the CNS. When we do so, it becomes possible for us

to migrate to a new contract for bugs to be fixed in our current contract. Consider the following code:

```
contract CNSContract {

    address providerAddress = msg.sender;

    mapping (bytes32 => address) contracts;



    modifier providerModifier() { if (msg.sender !=
providerAddress) throw; _; }

    function setContractFunction(bytes32 _nameOfContract, address
_contract) providerModifier {

         contracts[_nameOfContract] = _contract;

    }



    function getContractAddress(bytes32 _nameOfContract) constant
returns(address) {

         return contracts[_nameOfContract];

    }

}
```

Bug fixes are only permitted for the service provider only.

Here, the provider has been assigned the value msg.sender during the creation of the instance of CNSContract :

```
address providerAddress = msg.sender;
```

We also have the following:

```
modifier providerModifier() { if (msg.sender != providerAddress)
throw; _; }

function setContractFunction(bytes32 _nameOfContract, address
_contract) providerModifier {
```

```
    contracts[_nameOfContract] = _contract;
}
```

Due to this logic, only the provider who initially deployed CNS can register the contracts with it. This means the contracts and the CNS registered in CNS are all provided by CNS provider.

The problem of past data

The CNS seems to be good at solving the problem of upgrading, but there are still some issues which are to be solved. The data from the previous versions presents a problem.

To make this clear, consider a situation where you are creating a contract named “Someone” that will hold a field called “name”, then upgrade it and add a field named “age”. This may seem to be an easy task, but it is difficult.

Consider the simple object diagram given below:

During the upgrade, when you see someone_v1 who is registered in the CNS, the data group, that is, SomeoneStruct_v1, that had been registered at the time of Someone_v1 in the past will be separated from the new contract and it will become lost.

Solution

There are various ways that can be used to provide a solution to this problem:

1. Set address of Someone_v1#fields in Someone_v1

This can be done in any general programming language, but it becomes hard

to pass the address of a mapping object in a distributed environment.

2. Move all the data stored in fields of Someone_v1 to Someone_v1

This is the best method when there are several records, but with the increase of data, a lot of writing will be needed. This means that the writes will consume a high coinsAmount of gas, which makes the method infeasible.

3. Add Someone_vs 0 in the form a field to Someone_vs 1

When using this method, you only must add the old version address for one field. The good thing about this is you will not incur any additional gas consumption.

Adding Field Contract

If we hold the address of Someone_v1 as some field of Someone_v1, then it will be possible for us to keep on accessing the old data in Someone_v1. However, for the case of a new function added in Someone_v1, it will be hard to access Someone_v1#name.

Fields are simply dedicated field contracts that have accessors (getters/setters) for all the fields because it is hard to tell the accessors that will become useful in the future.

Suppose you have Someone_vs 1 inheriting from Someone_vs 0. After compilation, the contract size will be that of the two contracts combined. Ethereum has a restriction in terms of the maximum capacity of a transaction that can be added to a block. If the since is very large, the deployment process will become impossible.

The measure that can be put in place to prevent this is by separating the logic from the interface.

Chapter 16

Crowdfunding

In most cases, a good idea consumes a lot of effort and funds. You may ask for a donor funding, but you may not succeed. Crowdfunding will help get out of such a situation. You set up a goal and the deadline for the goal to be attained. If you miss the goal, the donations will be returned, therefore, the donors' risk will be reduced. The code is open and auditable, meaning there is no need for a trusted and centralized platform, and everyone will only be required to pay for the gas fees.

Tokens and DAOs

In this chapter, we will be making a better crowdfunding by solving two main problems which include how rewards are kept and managed and how the money is spent once the funds have been raised.

In crowdfunding, rewards are handled by a central unchangeable database that tracks all the donors. Anyone who misses the deadline for the campaign can't get in anymore and any donor who changed their mind cannot get out. However, we will just do this in a decentralized manner and create a token that will track the rewards. After contributing, one will get a token that they are able to trade, sell or even keep for a later use. When it comes a time to give the physical reward, the producer will only have to exchange the token for a real product. Donors will be able to keep their tokens even when the project fails to achieve its goals.

Also, those who give the funds will not have a say on how the money should be sent once it has been raised. In case a mismanagement occurs, the project

will not deliver anything at all. A Democratic Organization will be used to approve any money that comes out of the system. This is known as a “crowd equity” or a “crowdsale” and it is essential such that in some cases, the token can reward itself, especially in cases where people have gathered to create a common public good. We will be using a supply of 100.

The following code helps us create the crowdsale:

```
pragma solidity ^0.4.16;


interface myToken {

    function transfer(address coinsReceiver, uint coinsAmount);

}


contract CrowdsaleContract {

    address public beneficiaryAddress;

    uint public  fundingGoal;

    uint public raisedAmount;

    uint public deadlineForCampaign;

    uint public price;

    myToken public reward;

    mapping(address => uint256) public balanceAmount;

    bool goalReached = false;

    bool saleClosed = false;


    event GoalReachedEvent(address recipient, uint
contributedAmount);

    event FundTransfer(address backerAddress, uint coinsAmount,
bool isContribution);
```

```
    /**
     *The Constructor function
     *
     * Setup the owner
     */
    function CrowdsaleContract(
        address addressIfSuccessful,
        uint goalInEthers,
        uint duration,
        uint etherCostPerToken,
        address rewardTokenAddress
    ) {
        beneficiaryAddress = addressIfSuccessful;
         fundingGoal = goalInEthers * 1 ether;
        deadlineForCampaign = now + duration * 1 minutes;
        price = etherCostPerToken * 1 ether;
        reward = myToken(rewardTokenAddress);
    }


    /**
     * The fallback function
     *
     * The function with no name is the default function called
whenever any person sends funds to a contract
     */
```

```
    function () payable {
        require(!saleClosed);
        uint coinsAmount = msg.value;
        balanceAmount[msg.sender] += coinsAmount;
        raisedAmount += coinsAmount;
        reward.transfer(msg.sender, coinsAmount / price);
        FundTransfer(msg.sender, coinsAmount, true);
    }


    modifier deadlineReached() { if (now >= deadlineForCampaign)
_; }


    /**
     * Check whether the goal was reached
     *
     * Checks whether the goal or time limit has been reached
then end the campaign
     */
    function goalCheckFunction() deadlineReached {
        if (raisedAmount >=  fundingGoal){
            goalReached = true;
            GoalReachedEvent(beneficiaryAddress, raisedAmount);
        }
        saleClosed = true;
    }
```

```
    /**
     * Withdraw funds
     *
     * Check to see whether the goal or time limit has been
reached, if so, and the funding goal has been reached,
     * send the entire coinsAmount to beneficiaryAddress. If the
goal has not been reached, every contributor can withdraw
     * the coinsAmount they had contributed.
     */
    function withdrawSafely() deadlineReached {
        if (!goalReached) {
            uint coinsAmount = balanceAmount[msg.sender];
            balanceAmount[msg.sender] = 0;
            if (coinsAmount > 0) {
                if (msg.sender.send(coinsAmount)) {
                    FundTransfer(msg.sender, coinsAmount, false);
                } else {
                    balanceAmount[msg.sender] = coinsAmount;
                }
            }
        }

        if (goalReached && beneficiaryAddress == msg.sender) {
            if (beneficiaryAddress.send(raisedAmount)) {
                FundTransfer(beneficiaryAddress, raisedAmount,
false);
```

```
            } else {

                //If we fail to send funds to beneficiaryAddress,
unlock the funders balance

                goalReached = false;

            }

        }

    }

}
```

Note how the ‘deadlineForCampaign” and “ fundingGoal” variables have been set in the “CrowdsaleContract” function:

```
 fundingGoal = goalInEthers * 1 ether;

deadlineForCampaign = now + duration * 1 minutes;

price = etherCostPerToken * 1 ether;
```

Those are some of the Solidity special keywords that can help you to code. The system measures the coinsAmount of ether in units known as Wei. The code given above converts the funding goal into Wei. This is achieved by multiplying it by 1,000,000,000,000,000,000. A timestamp is then created which is X minutes away from today by use of a combination of special keywords “now” and “minutes”. The contract given below instantiates a contract at a given address:

```
reward = myToken(rewardTokenAddress);
```

The contract knows what a token is since it had been defined earlier when we started the code as follows:

contract myToken { function transfer(address coinsReceiver, uint coinsAmount){ } }

However, this has not described how the contract works or all the functions that it has, but it only describes the functions the contract needs. A token is simply a contract with a transfer function.

How to Use

Begin by deploying the contract. Add the address of the contract that you have created to the field “if successful, send to”. Add 250 ethers as the funding goal. If you are just testing, you can set the crowdfunding duration to 3-10 minutes, if you are really in need of raising funds, put it to a larger coinsAmount such as 45,000 (31 days). The cost of ether for every token should be calculated depending on the number of tokens you are putting up for sale. In this example, just use 5 ethers.

The address of the created token should be added to ‘token reward address”. Pu the gas price, click deploy then wait for the crowdsale to be created. After the creation of the crowdsale page, you should deposit enough rewards for the rewards to be paid back. Click on the address of the crowdsale, deposit and send 50 gadgets to the crowdsale.

Note that we are building a crowdsale that will be under the control of token holders completely. However, this is dangerous in that if someone controls 50%+1 of all the tokens, they can send all the funds to themselves. You can create a simple code on the association contract that will prevent these hostile takeovers or have all the funds send to some simple address. To make it simple, we will be selling half of the gadgets, and if you need of decentralizing this further, you can split the remaining half between trusted organizations.

Raise funds

Once your crowdsale has all the tokens that are necessary, you can easily contribute to it and do it from any Ethereum wallet by just sending funds to it. The relevant code bit is given below:

```
function () {
```

```
    require(!saleClosed);

    uint coinsAmount = msg.value;

    // ...
```

The unnamed function will be the default function that will be executed any time a contract has received ether. The function will automatically check whether the crowdsale is active, calculate the number of tokens the caller bought, and it will send the equivalent of these. In case the crowdsale has ended or the contract runs out of tokens, the contract will throw, meaning that the execution will stop, and the ether sent will be returned. However, all the gas will be spent.

This is good as it helps one prevent landing into a situation where they will be left without their ether or tokens. Previously, we could self-destruct the contract once the crowdsale has ended, meaning that any transactions that is sent after the moment will lose their funds. When a fallback function that throws once a sale is over is created, the possibility of anyone losing their money will be prevented.

The contract has the selfWithdrawal() function with no parameters, and this can be executed by the beneficiary to access the coinsAmount that has been raised or by the funders to get back their funds, in any case, a failed fundraise occurs.

In our code, the crowdsale can reach its targets or it may not. Since we have a limited coinsAmount of tokens, it means that after reaching the global, no one else will be able to contribute. However, the history of crowdfunding has so many projects that overshoot their goals in a less time than predicted.

Unlimited Crowdsale

We need to modify our project in such a way that instead of sending a limited coinsAmount of tokens, the project will create a new token out of thin air anytime someone sends them ether. We should begin by creating a Mintable

token.

After that, the crowdsale should be modified by changing all the “transfer” mentions to “mintTokenFunction”. This is shown below:

```
contract myToken { function mintTokenFunction(address
coinsReceiver, uint coinsAmount){  } }

// ...

    function () {

        // ...

        reward.mintTokenFunction(msg.sender, coinsAmount /
price);

        // ...

    }
```

After publishing the crowdsale contract, you should get its address then go to your Token Contract and execute a Change Ownership function. This way, the crowdsale will be able to call the Mint Token as many times as it wants.

However, you should note that this will expose you to the risk of hostile takeover. During the crowdsale, in case anybody contributes more coinsAmount than the total contributed by the others, they will be able to exercise control over the whole pie and even steal it. You must investigate a way of controlling this. The following are some of the mechanisms that can be put into place to control this:

1. Modify the crowdsale in such a way that once a token has been bought, it will also send a similar coinsAmount of tokens to the account of the founder so that they control only 50% of the project.

2. Change the organization to create veto power to a trusted third party capable of stopping any hostile proposal.

3. Change the token to permit a central trusted party to require a

verification that there is no single entity capable of controlling the majority of them.

Call Scheduling

In Ethereum, contracts are passive, meaning that they are only capable of doing something after activating them. However, there are third-party community services that have been developed to provide us with such services. An example of this is Ethereum Alarm Clock, which is an open marketplace in which anyone can receive ether to execute scheduled calls or even pay ether to schedule them. We will be using version 0.6.0 of this app.

First, the contract should be added to the watchlist. Go to the “Contracts” tab then “Watch contract” instead of “deploy contract”. You can name it “Ethereum Alarm Clock”. Use the address “0xe109EcB193841aF9dA3110c80FDd365D1C23Be2a”. The icon should now appear as a green-eyed creature. Add the code given below as the JSON interface:

```
[
   {
      "constant":false,
      "inputs":[
         {
            "name":"contAddress ",
            "type":"address"
         },
         {
            "name":"signature",
            "type":"bytes4"
```

```
        },
        {
            "name":"targetABlock",
            "type":"uint256"
        }
    ],
    "name":"callSchedule",
    "outputs":[
        {
            "name":"",
            "type":"address"
        }
    ],
    "type":"function"
},
{
    "constant":false,
    "inputs":[
        {
            "name":"contAddress ",
            "type":"address"
        },
        {
            "name":"signature",
            "type":"bytes4"
```

```
        },
        {
            "name":"targetABlock",
            "type":"uint256"
        },
        {
            "name":"proposedGas",
            "type":"uint256"
        },
        {
            "name":"gracePeriod",
            "type":"uint8"
        }
    ],
    "name":"callSchedule",
    "outputs":[
        {
            "name":"",
            "type":"address"
        }
    ],
    "type":"function"
},
{
    "constant":true,
```

```
    "inputs":[

    ],
    "name":"getDefaultPayment",
    "outputs":[
       {
          "name":"",
          "type":"uint256"
       }
    ],
    "type":"function"
},
{
    "constant":true,
    "inputs":[

    ],
    "name":"getDefaultFee",
    "outputs":[
       {
          "name":"",
          "type":"uint256"
       }
    ],
    "type":"function"
```

```
},
{
   "constant":false,
   "inputs":[
      {
         "name":"contAddress ",
         "type":"address"
      },
      {
         "name":"signature",
         "type":"bytes4"
      },
      {
         "name":"targetABlock",
         "type":"uint256"
      },
      {
         "name":"proposedGas",
         "type":"uint256"
      }
   ],
   "name":"callSchedule",
   "outputs":[
      {
         "name":"",
```

```
            "type":"address"
         }
      ],
      "type":"function"
   },
   {
      "constant":true,
      "inputs":[
         {
            "name":"addressForCall",
            "type":"address"
         }
      ],
      "name":"nextSiblingCall",
      "outputs":[
         {
            "name":"",
            "type":"address"
         }
      ],
      "type":"function"
   },
   {
      "constant":true,
      "inputs":[
```

```
        {
            "name":"addressForCall",
            "type":"address"
        }
    ],
    "name":"knownCall",
    "outputs":[
        {
            "name":"",
            "type":"bool"
        }
    ],
    "type":"function"
},
{
    "constant":true,
    "inputs":[
        {
            "name":"basePay",
            "type":"uint256"
        }
    ],
    "name":"lowestCallCost",
    "outputs":[
        {
```

```
            "name":"",
            "type":"uint256"
         }
      ],
      "type":"function"
   },
   {
      "constant":false,
      "inputs":[
         {
            "name":"contAddress ",
            "type":"address"
         },
         {
            "name":"signature",
            "type":"bytes4"
         },
         {
            "name":"targetABlock",
            "type":"uint256"
         },
         {
            "name":"proposedGas",
            "type":"uint256"
         },
```

```
      {
         "name":"gracePeriod",
         "type":"uint8"
      },
      {
         "name":"basePay",
         "type":"uint256"
      }
   ],
   "name":"callSchedule",
   "outputs":[
      {
         "name":"",
         "type":"address"
      }
   ],
   "type":"function"
},
{
   "constant":true,
   "inputs":[

   ],
   "name":"lowestCallCost",
   "outputs":[
```

```
      {
         "name":"",
         "type":"uint256"
      }
   ],
   "type":"function"
},
{
   "constant":false,
   "inputs":[
      {
         "name":"contAddress ",
         "type":"address"
      },
      {
         "name":"signature",
         "type":"bytes4"
      },
      {
         "name":"targetABlock",
         "type":"uint256"
      },
      {
         "name":"proposedGas",
         "type":"uint256"
```

```
      },
      {
         "name":"gracePeriod",
         "type":"uint8"
      },
      {
         "name":"basePay",
         "type":"uint256"
      },
      {
         "name":"baseFee",
         "type":"uint256"
      }
   ],
   "name":"callSchedule",
   "outputs":[
      {
         "name":"",
         "type":"address"
      }
   ],
   "type":"function"
},
{
   "constant":true,
```

```
    "inputs":[
       {
          "name":"basePay",
          "type":"uint256"
       },
       {
          "name":"baseFee",
          "type":"uint256"
       }
    ],
    "name":"lowestCallCost",
    "outputs":[
       {
          "name":"",
          "type":"uint256"
       }
    ],
    "type":"function"
},
{
    "constant":true,
    "inputs":[

    ],
    "name":"getLowestCallGas",
```

```
    "outputs":[
       {
          "name":"",
          "type":"uint256"
       }
    ],
    "type":"function"
},
{
    "constant":true,
    "inputs":[

    ],
    "name":"getWindowSizeForCall",
    "outputs":[
       {
          "name":"",
          "type":"uint256"
       }
    ],
    "type":"function"
},
{
    "constant":true,
    "inputs":[
```

```
      {
         "name":"blockNumber",
         "type":"uint256"
      }
   ],
   "name":"nextCall",
   "outputs":[
      {
         "name":"",
         "type":"address"
      }
   ],
   "type":"function"
},
{
   "constant":true,
   "inputs":[

   ],
   "name":"getShortestGracePeriod",
   "outputs":[
      {
         "name":"",
         "type":"uint256"
      }
```

```
        ],
        "type":"function"
    }
]
```

For those on a test net, you should use “0xb8Da699d7FB01289D4EF718A55C3174971092BEf” as the address.

Now, click the green icon that you added and select a function call under “Write to contract” title. You will have multiple “Schedule Call” functions. Select the first one that has three fields only:

1. contAddress - this is the address of the crowdsale contract that you have deployed.

2. the signature should be “0x01cb3b20”. You can get the signature for a function by running it on the confirmation window. Instead of typing the password, copy the code in Data field. The signature for the function will be the first 10 characters that are written in bold.

3. targetBlock should be the block number where you need the function to be executed.

In the crowdsale contract, the deadlineForCampaign is specified using a timestamp. However, the Alarm clock schedules the calls based on the block numbers. The Ethereum’s block time is about 17 seconds, and there is a need for us to calculate the block number that will be past the deadline probabilistically. To do this, we should use the formula given below:

block_number + duration (In minutes) * 60 / 17 + buffer

Where buffer is the number of blocks that is large enough and you can rely on it occurring after a crowdsale deadline. For the case of short crowdsales with a duration of less than a day, a buffer of about 200 blocks will be sufficient. If the crowdsale takes a duration of about 30 days, then you should choose a number close to 5000.

The following chart can help you get a rough estimate of blocks you should add to a current block when calculating the targetBlock:

- 60 minutes duration (1 hour): 212 blocks
- 1440 minutes duration (1 day): 5082 blocks
- 10,800 minutes duration (1 week): 38,117 blocks
- 44,640 minutes duration (1 month): 157,553 blocks

On the field for “send”, you should send ether that is enough for you to pay the transaction fee, and more to be used for paying the scheduler. Any extra money that you send will be refunded. This means that it will be good for you to send at least 0.25 ether to stay on the safe side.

Once done, press execute, and the call will be scheduled. Note that there is no guarantee that someone will execute it, so it will be good for you to check once the deadline has expired to be sure.

Chapter 17

Decentralized Autonomous Organization

We have created contracts that are owned and executed by the other accounts, and these accounts are owned by individuals. However, in the Ethereum blockchain, there is no discrimination against humans or robots and contracts can create arbitrary actions similarly to any other account. Contracts can participate in crowdsales, own tokens and even act as voting members of the other contracts.

In this chapter, we will be building a democratic and decentralized organization existing solely on the blockchain, and it will be capable of doing anything that an account can do. The organization will have a central manager responsible for deciding who the members are as well as the voting rules, but this can be changed.

Our goal in this chapter is to introduce democracy on the blockchain. This is the concept used to introduce democracy in an organization that has an Owner working like the CEO, President or Administrator. The Owner can add or remove voting members to the organization.

Any member can make a proposal in the form of Ethereum transaction to send ether or execute a contract and the other members will be able to vote in support or against this. After a predetermined period and when a number of the members have voted, it will be possible to execute the proposal. The contract will count the votes and in case there are enough votes, the given transaction will be executed. Here is the code:

```
pragma solidity ^0.4.16;
```

```
contract owned {

    address public ownerAddress;


    function owned()  public {

        ownerAddress = msg.sender;

    }


    modifier onlyOwnerModifier {

        require(msg.sender == ownerAddress);

        _;

    }


    function transferOwnership(address newOwner)
 onlyOwnerModifier  public {

        ownerAddress = newOwner;

    }

}


contract tokenRecipient {

    event etherReceivedEvent(address sender, uint coinsAmount);

    event tokensReceivedEvent(address _from, uint256 _value,
address _token, bytes _extraData);


    function approveReceipt(address _from, uint256 _value,
address _token, bytes _extraData) public {

        Token tok = Token(_token);
```

```
        require(tok.transferSource(_from, this, _value));
        tokensReceivedEvent(_from, _value, _token, _extraData);
    }

    function () payable  public {
        etherReceivedEvent(msg.sender, msg.value);
    }
}

interface Token {
    function transferSource(address _from, address _to, uint256
_value) public returns (bool successful);
}

contract CongressContract is owned, tokenRecipient {
    // Variables and events for the contract
    uint public lowestQuorum;
    uint public debatePeriod;
    int public marginOfMajority;
    Proposal[] public proposals;
    uint public numberOfProposals;
    mapping (address => uint) public idOfMember;
    Member[] public availableMembers;

    event ProposalCreated(uint idOfProposal, address recipient,
uint coinsAmount, string description);
```

```
    event ReadilyVoted(uint idOfProposal, bool position, address
voterAddress, string justificationString);

    event TalliedProposal(uint idOfProposal, int result, uint
quorum, bool active);

    event ChangedMembership(address memberAddress, bool
isMemberBool);

    event rulesChange(uint newLowestQuorum, uint newDebatePeriod,
int newMarginOfMajority);


    struct Proposal {

        address recipient;

        uint coinsAmount;

        string description;

        uint deadlineofVoting;

        bool executed;

        bool passedProposal;

        uint totalVotes;

        int obtainedResult;

        bytes32 hashOfProposal;

        Vote[] votes;

        mapping (address => bool) voted;

    }


    struct Member {

        address memberAddress;

        string name;

        uint memberFrom;
```

```
    }


    struct Vote {
        bool isSupporting;
        address voterAddress;
        string justificationString;
    }


    // Modifier to allows only shareholders to create new
proposals and vote
    modifier onlyMembersModifier {
        require(idOfMember[msg.sender] != 0);
        _;
    }


    /**
     * The constructor function
     */
    function CongressContract (
        uint minimumQuorumForProposals,
        uint minutesForDebate,
        int marginOfVotesForMajority
    )  payable public {
        changeVotingRules(minimumQuorumForProposals,
minutesForDebate, marginOfVotesForMajority);
        // It is good to add first member as empty
```

```
        createMember(0, "");
        // then we add the founder, to save a step later
        createMember(ownerAddress, 'founder');
    }

    /**
     * Add a member
     *
     * Make the `targetMemberAddress` to be a member named `memberName`
     *
     * @param targetMemberAddress ethereum address which is to be added
     * @param memberName public name for the member
     */
    function  createMember(address targetMemberAddress, string memberName)  onlyOwnerModifier public {
        uint id = idOfMember[targetMemberAddress];
        if (id == 0) {
            idOfMember[targetMemberAddress] = availableMembers.length;
            id = availableMembers.length++;
        }

        availableMembers[id] = Member({memberAddress: targetMemberAddress, memberFrom: now, name: memberName});
        ChangedMembership(targetMemberAddress, true);
    }
```

```
    /**
     * Remove member
     *
     * @notice Remove the membership from `targetMemberAddress`
     *
     * @param targetMemberAddress ethereum address which is to be
removed
     */
    function deleteMember(address targetMemberAddress)
 onlyOwnerModifier public {
        require(idOfMember[targetMemberAddress] != 0);

        for (uint i = idOfMember[targetMemberAddress];
i<availableMembers.length-1; i++){
            availableMembers[i] = availableMembers[i+1];
        }
        delete availableMembers[availableMembers.length-1];
        availableMembers.length--;
    }

    /**
     * Change the voting rules
     *
     * Change so that the proposals will need to be discussed for
not less
     * than `minutesForDebate/60` hours,
```

```
     * It should have at least `minimumQuorumForProposals` votes, and have
     * 50% + `marginOfVotesForMajority` votes in order to be executed
     *
     * @param minimumQuorumForProposals how many members must vote on a proposal for it to be executed
     * @param minutesForDebate the minimum delay between the time a proposal is made and when it may be executed
     * @param marginOfVotesForMajority the proposal should have 50% plus this number
     */
    function changeVotingRules(
        uint minimumQuorumForProposals,
        uint minutesForDebate,
        int marginOfVotesForMajority
    )  onlyOwnerModifier public {
        lowestQuorum = minimumQuorumForProposals;
        debatePeriod = minutesForDebate;
        marginOfMajority = marginOfVotesForMajority;

        rulesChange(lowestQuorum, debatePeriod, marginOfMajority);
    }

    /**
     * Add a Proposal
     *
```

```
     * Propose to send the `amountOfWei / 1e18` ether to the
`beneficiaryAddress` for `jobDescription`. The
`bytecodeOfTransaction ? Contains : It does not contain` code.
     *
     * @param beneficiaryAddress to send ether to
     * @param amountOfWei ether coinsAmount to be send, in wei
     * @param jobDescription Description of job
     * @param bytecodeOfTransaction bytecode of transaction
     */
    function createNewProposal(
        address beneficiaryAddress,
        uint amountOfWei,
        string jobDescription,
        bytes bytecodeOfTransaction
    )
        onlyMembersModifier public
        returns (uint idOfProposal)
    {
        idOfProposal = proposals.length++;
        Proposal storage prop = proposals[idOfProposal];
        prop.recipient = beneficiaryAddress;
        prop.coinsAmount = amountOfWei;
        prop. jobDescription = jobDescription;
        prop.hashOfProposal = keccak256(beneficiaryAddress,
amountOfWei, bytecodeOfTransaction);
        prop.deadlineofVoting = now + debatePeriod * 1 minutes;
        prop.executed = false;
```

```
        prop.passedProposal = false;

        prop.totalVotes = 0;

        ProposalCreated(idOfProposal, beneficiaryAddress,
amountOfWei, jobDescription);

        numberOfProposals = idOfProposal+1;



        return idOfProposal;

    }



    /**

     * Add proposal in Ether

     *

     * Propose to send `etherAmount` ether to
`beneficiaryAddress` for `jobDescription`. `bytecodeOfTransaction
? Contains : Does not contain` code.

     * This is a convenience function to use if the coinsAmount
to be given is in round number of ether units.

     *

     * @param beneficiaryAddress who to send the ether to

     * @param etherAmount coinsAmount of ether to send

     * @param jobDescription Description of job

     * @param bytecodeOfTransaction bytecode of transaction

     */

    function newProposal(

        address beneficiaryAddress,

        uint etherAmount,

        string jobDescription,
```

```
        bytes bytecodeOfTransaction
    )
        onlyMembersModifier public
        returns (uint idOfProposal)
    {
        return createNewProposal(beneficiaryAddress, etherAmount
* 1 ether, jobDescription, bytecodeOfTransaction);
    }

    /**
     * Check whether the proposal code matches
     *
     * @param proposalNumber ID number of proposal to query
     * @param beneficiaryAddress to send ether to
     * @param amountOfWei the coinsAmount of ether to be send
     * @param bytecodeOfTransaction bytecode of transaction
     */
    function checkCode(
        uint proposalNumber,
        address beneficiaryAddress,
        uint amountOfWei,
        bytes bytecodeOfTransaction
    )
        constant public
        returns (bool checkOut)
    {
```

```
        Proposal storage prop = proposals[proposalNumber];

        return prop.hashOfProposal ==
keccak256(beneficiaryAddress, amountOfWei,
bytecodeOfTransaction);

    }



    /**

     * Log a vote for the proposal

     *

     * Vote `inSupportOfProposal? in support of : against`
proposal #`proposalNumber`

     *

     * @param proposalNumber number of proposal

     * @param inSupportOfProposal either in favor or against it

     * @param justificationString an optional justificationString
text

     */

    function vote(

        uint proposalNumber,

        bool inSupportOfProposal,

        string justificationString

    )

        onlyMembersModifier public

        returns (uint voteID)

    {

        Proposal storage prop = proposals[proposalNumber];   //
Getting proposal

        require(!prop.voted[msg.sender]);  // If already voted,
```

```
cancel

        prop.voted[msg.sender] = true;   // Set the voter as
having voted

        prop.totalVotes++;           // Increase number of votes

        if (inSupportOfProposal) {     // If they are in support
of the proposal

            prop.obtainedResult++;     // Increase the score

        } else {                    // If they don't support

            prop.obtainedResult--;    // Decrease the score

        }



        // Create a log for the event

        ReadilyVoted(proposalNumber,  inSupportOfProposal,
msg.sender, justificationString);

        return prop.totalVotes;

    }



    /**

     * Finish vote

     *

     * Count the votes proposal #`proposalNumber` and execute it
if approved

     *

     * @param proposalNumber proposal number

     * @param bytecodeOfTransaction optional: if the transaction
contained a bytecode, you need to send it

     */

    function runProposal(uint proposalNumber, bytes
```

```
bytecodeOfTransaction) public {

        Proposal storage prop = proposals[proposalNumber];



        require(now > prop.deadlineofVoting
                     // If it's past voting deadlineForCampaign

            && !prop.executed
                    // and it hasn't been executed

            && prop.hashOfProposal == keccak256(prop.recipient,
prop.coinsAmount, bytecodeOfTransaction)  // and supplied code
matches the proposal

            && prop.totalVotes >= lowestQuorum);
                 // and the minimum quorum has been reached...



        // ...execute the result



        if (prop.obtainedResult > marginOfMajority) {

            // Proposal has been passed; execute the transaction



            prop.executed = true; // Avoid a recursive calling

            require(prop.recipient.call.value(prop.coinsAmount)
(bytecodeOfTransaction));



            prop.passedProposal = true;

        } else {

            // Proposal has failed

            prop.passedProposal = false;

        }
```

```
        // Fire the Events

        TalliedProposal(proposalNumber, prop.obtainedResult,
prop.totalVotes, prop.passedProposal);

    }

}
```

Deployment

Open the wallet then go to Contracts tab and press “deploy contract”. In the box for “solidity code”, paste all the code given above. On contract picker, select “CongressContract” and all the setup variables will be shown. They include the following:

1. The minimum quorum for proposals- this is the minimum number of votes that a proposal should have to be executed.

2. Minutes for debate- this is the minimum time that should pass for it to be executed.

3. The margin of votes for the majority- for a proposal to pass, it must have 50% of votes plus a margin. For a simple majority, you can leave it at 0. If you need an absolute consensus, you should put it at “number of members – 1”.

Note that it is possible for you to change the above parameters later. Just choose a name, set debate time to 5 minutes then leave the rest at 0. The estimate cost of the ether will be displayed at the lower part of the page. If you need to save, you can lower the price. However, if you do this, you may have to wait for long to have the contract created. Click “Deploy” then type in the password and wait.

After some seconds, the dashboard will be opened. Just scroll to its bottom and you will see that your contract has been created. You can click on the name of the contract to see it. Whenever you need to get this contract, you

only have to open the “Contracts” tab.

Sharing the DAO

For you to be able to share your DAO with the others, they should have both the interface file and the contract address. The interface file is simply a small text string that can be used as an instruction manual for the contract. Click “copy address” in order to get the former, then click “show interface” in order to reveal the latter.

On another computer, open the Contracts tab then click “watch contract”. Add the right address and the interface then click “OK”.

Interaction with the Contract

All the functions that you can execute for free on the contract will be shown in the section for “read from contract”. This is because the functions will only be reading the information from the blockchain. You will be able to see current “owner” of a contract. Note that the owner, in this case, should be the account that did upload the contract.

On “Write to contract”, you will see the list of functions that will try to perform a computation to save data to the blockchain, and this will cost you some ether. Choose “createNewProposal” and all the options for the functions will be shown.

Before you can begin with the contract, you should add some new members who will be able to vote. On the picker for “select function”, click “Add Member”. Type the address of the individual whom you need to make a member. If you need to remove a member, you should choose “Remove Member”. On “execute from”, ensure that you have the same account that was set as owner since this action can only be executed by the main administrator. Press “execute” then wait for some seconds for next block to

go through with the change.

Note that you will not get the members' list, but it is possible for you to check whether one is a member by adding their address on Members function on the column for "Read from contract".

Also, if you need the contract to have some money of its own, you should deposit either ether or another token into it to avoid having a toothless organization. More to the top right corner then press "Transfer Ether & Tokens".

Adding a Simple Proposal

It is now time for us to add the first proposal to the contract. Choose "New Proposal" on function picker.

In the field for "beneficiaryAddress", enter the name of the person to whom you need to send ether, then type the coinsAmount of ether you need to send on the field for "etherAmount". Note that the coinsAmount must be an integer. Lastly, enter a description as to why you need to send the ether. Leave the field for "transactionByteCode" blank. Click execute then enter your password. After some seconds, you will see the numberOfProposals increase to 1, while the first proposal, which is number 0 will be shown in the left column. As more proposals are added, you will be able to see any proposal by entering its number in the "proposals" field and you will be able to read more about it.

It will also be simple for you to vote on a proposal. You only have to move to function picker then choose "Vote". On the first box, type the number of the proposal, then check "Yes" box to agree with it. If you need to vote against the proposal, just leave it blank. Next, click "execute" in order to send your vote.

After the expiry of the voting time, you can select "runProposal". If you had only created a transaction for sending ether only, then the field for

“bytecodeOfTransaction” can be left blank. Once you have pressed “execute” and before you can type the password, a new screen will appear, so pay attention to it.

If you get a warning on the field for “estimated fee consumption”, it indicates that the function that has been called will not be executed but it will instead be terminated. There are many causes of this, but in our case, this will occur if you execute the contract before passing of the deadlineForCampaign, or if the user needs to send different bytecode data that what was contained in the original proposal. In case any of these happens, then the execution of the contract will be terminated and the user who had tried the illegal transaction will lose all ether sent to pay the transaction fees.

In case the transaction executed successfully, then you will be able to see the results after some time. The “executed” will change to true and the right coinsAmount of ether will be subtracted from the balance of the contract and added to the address of the recipient.

Creating a Complex Proposal

The same democracy can be used for executing any transaction on the Ethereum, provided you can figure out the bytecode generated by the transaction.

In this section, we will demonstrate the fact that the contract can be used for holding more than ether and it can do transactions and it can perform transactions in any asset that is based on Ethereum. Begin by creating a token belonging to one of the normal accounts. Open the contract page, then click “Transfer Ether & Tokens” in order to transfer them to the new Congress contract. To keep things simple, avoid sending more than half of the coins to your DAO. After that, we will simulate the action that we need to execute. If you need to propose the new DAO to send 500mg of gold token to an individual as a payment, follow similar steps you would follow to execute the transaction from an account that you own then press “send”. Once you see the confirmation screen, do not type the password.

Instead of typing the password, click the link for “SHOW RAW DATA” then copy the data that is displayed on “RAW DATA” field then save it to a notepad or text file. You can then cancel the transaction. The address of the contract that you will be calling for the operation will also be needed, which is the token contract in our case. This can be found on the tab for “Contracts”, so save it somewhere.

You can then create a new proposal on the congress contract and it should have the parameters given below:

- You are the beneficiary, so add the address of your token. Be attentive if it is a similar icon.

- Leave the field for “Ether coinsAmount” blank.

- In the field for “Job description”, write a small description of what you need to achieve.

- In the field for “Transaction Bytecode”, save the bytecode that you saved from the data field during the previous step.

After some seconds, you will be able to see the details of your proposal. The transaction bytecode will not be shown, but you will only have the “transaction hash”. The bytecode can be long and expensive for us to store on the blockchain. Instead of having to archive it, the individual executing the call later may provide the bytecode.

However, this leads to a loophole as far as security is concerned. How can the proposal be voted when the code behind it is not there? The transaction hash comes in here. Scroll on “read from contract” function list then you will find a proposal checker function, in which anyone can put the function parameters and check whether they match the one that is being voted on. With this, there is a guarantee that proposals will not be executed unless the bytecode hash matches exactly the one that is provided in the code.

Anyone will be able to easily check the proposal by following similar steps to get the right bytecode then add the number of the proposal and the other parameters to the function named “Check proposal code” and located below

“Read from contract”.

The other part of the voting process will be the same. The members can vote and once the deadline expires, one will be able to execute the proposal. The difference is that this time you will have to provide a bytecode which is like the one you had provided before. Warnings are normally displayed on the confirmation window, so pay attention to them. In case you are told the code will not be executed, just check to see whether the deadline has passed, whether there are enough votes and if the transaction bytecode checks out.

Shareholder Association

Our previous contract acts like an invitation club only, in which the members are invited or barred by the authority of the president. However, there are several limitations associated with this. What will happen if one needs to change his or her address? What will happen if some members are having more weight than the others? What can you do if you need to sell your shares or membership in an open market? What if you need the organization to work as a constant decision machine by the shareholders?

We should modify our contract a bit to connect it to a particular token, and this will work as the holding shares for the contract. Our first step should be to create this token.

In the Ethereum ecosystem, tokens can be used to represent any tradable good such as coins, gold certificates, loyalty points, in-game items, IOUs etc. Since the tokens are used to implement the basic features in a particular way, this is an indication that the token will be compatible with the Ethereum wallet as well as any contract or client using similar standards. Go to the chapter for “Creating a Token” and learn how to create a token. When creating your token, ensure that it has an initial supply of 100, decimals of 0 and the percentage sign (%) as the symbol. If you need to be able to trade in fractions of a percentage, increase the supply by 100x or 1000x, then ensure that you add the corresponding number of zeros as decimals. Deploy the contract then save its address to a text file. The shareholder code is given

below:

```
pragma solidity ^0.4.16;


contract owned {

    address public ownerAddress;


    function owned() {

        ownerAddress = msg.sender;

    }


    modifier  onlyOwnerModifier {

        require(msg.sender == ownerAddress);

        _;

    }


    function transferOwnership(address newOwner)
 onlyOwnerModifier {

        ownerAddress = newOwner;

    }

}


contract tokenRecipient {

    event etherReceivedEvent(address sender, uint coinsAmount);

    event tokensReceivedEvent(address _from, uint256 _value,
address _token, bytes _extraData);
```

```
    function approveReceipt(address _from, uint256 _value,
address _token, bytes _extraData){

        Token tok = Token(_token);

        require(tok.transferSource(_from, this, _value));

        tokensReceivedEvent(_from, _value, _token, _extraData);

    }


    function () payable {

        etherReceivedEvent(msg.sender, msg.value);

    }

}


contract Token {

    mapping (address => uint256) public balanceAmount;

    function transferSource(address _from, address _to, uint256
_value) returns (bool successful);

}


/**

 * Shareholder association contract

 */

contract ShareholderAssociation is owned, tokenRecipient {


    uint public lowestQuorum;

    uint public debatePeriod;

    Proposal[] public proposals;
```

```
    uint public numberOfProposals;

    Token public tokenAddressForShares;


    event ProposalCreated(uint idOfProposal, address recipient,
uint coinsAmount, string description);

    event ReadilyVoted(uint idOfProposal, bool position, address
voterAddress);

    event TalliedProposal(uint idOfProposal, uint result, uint
quorum, bool active);

    event rulesChange(uint newLowestQuorum, uint newDebatePeriod,
address newTokenAddressForShares);


    struct Proposal {

        address recipient;

        uint coinsAmount;

        string description;

        uint deadlineofVoting;

        bool executed;

        bool passedProposal;

        uint totalVotes;

        bytes32 hashOfProposal;

        Vote[] votes;

        mapping (address => bool) voted;

    }


    struct Vote {

        bool isSupporting;
```

```
        address voterAddress;
    }

    // Modifier allowing only shareholders to vote as well as
create new proposals
    modifier onlyShareholdersModifier{
        require(tokenAddressForShares.balanceAmount(msg.sender) >
0);
        _;
    }

    /**
     * The constructor function
     *
     * First time setup
     */
    function ShareholderAssociation(Token sharesAddress, uint
minimumNumberOfSharesForAVoteToPass, uint minutesForDebate)
payable {
        changeVotingRules(sharesAddress,
minimumNumberOfSharesForAVoteToPass, minutesForDebate);
    }

    /**
     * Change the voting rules
     *
     * Make proposals so that need to be discussed for not less
than `minutesForDebate/60` hours
```

```
     * and all the voters combined must have more than
`minimumNumberOfSharesForAVoteToPass` shares of token
`sharesAddress` in order to be executed

     *

     * @param sharesAddress the token address

     * @param minimumNumberOfSharesForAVoteToPass proposal will
vote only if sum of held shares by all the voters exceed this
number

     * @param minutesForDebate the minimum delay between the time
a proposal is made and the time it can be executed

     */

    function changeVotingRules(Token sharesAddress, uint
minimumNumberOfSharesForAVoteToPass, uint minutesForDebate)
 onlyOwnerModifier {

        tokenAddressForShares = Token(sharesAddress);

        if (minimumNumberOfSharesForAVoteToPass == 0 )
minimumNumberOfSharesForAVoteToPass = 1;

        lowestQuorum = minimumNumberOfSharesForAVoteToPass;

        debatePeriod = minutesForDebate;

        rulesChange(lowestQuorum, debatePeriod,
tokenAddressForShares);

    }



    /**

     * Add the Proposal

     *

     * Propose to send the `amountOfWei / 1e18` ether to the
`beneficiaryAddress` for `jobDescription`. `bytecodeOfTransaction
? Contains : Doesn’t have` code.

     *

     * @param beneficiaryAddress to send ether to
```

```
     * @param amountOfWei coinsAmount of ether to send, measured
in wei

     * @param jobDescription the job description

     * @param bytecodeOfTransaction the bytecode of the
transaction

     */

    function createNewProposal(

        address beneficiaryAddress,

        uint amountOfWei,

        string jobDescription,

        bytes bytecodeOfTransaction

    )

        onlyShareholders

        returns (uint idOfProposal)

    {

        idOfProposal = proposals.length++;

        Proposal storage prop = proposals[idOfProposal];

        prop.recipient = beneficiaryAddress;

        prop.coinsAmount = amountOfWei;

        prop.description = jobDescription;

        prop.hashOfProposal = sha3(beneficiaryAddress,
amountOfWei, bytecodeOfTransaction);

        prop.deadlineofVoting = now + debatePeriod * 1 minutes;

        prop.executed = false;

        prop.passedProposal = false;

        prop.totalVotes = 0;

        ProposalCreated(idOfProposal, beneficiaryAddress,
```

```
amountOfWei, jobDescription);
        numberOfProposals = idOfProposal+1;

        return idOfProposal;
    }

    /**
     * Add a proposal in Ether
     *
     * Propose to send the `etherAmount` ether to the
`beneficiaryAddress` for the `jobDescription`.
`bytecodeOfTransaction ? Contains : Doesn't have ` code.

     *
     * @param beneficiaryAddress individual to send ether to
     * @param etherAmount the coinsAmount of ether to be send
     * @param jobDescription Description of the job
     * @param bytecodeOfTransaction the bytecode of the
transaction
     */
    function newProposal(
        address beneficiaryAddress,
        uint etherAmount,
        string jobDescription,
        bytes bytecodeOfTransaction
    )
        onlyShareholders
```

```
        returns (uint idOfProposal)
    {
        return createNewProposal(beneficiaryAddress, etherAmount
* 1 ether, jobDescription, bytecodeOfTransaction);
    }

    /**
     * Check whether the proposal code matches
     *
     * @param proposalNumber the ID number of proposal to query
     * @param beneficiaryAddress who to send the ether to
     * @param amountOfWei coinsAmount of ether to send
     * @param bytecodeOfTransaction bytecode of transaction
     */
    function checkCode(
        uint proposalNumber,
        address beneficiaryAddress,
        uint amountOfWei,
        bytes bytecodeOfTransaction
    )
        constant
        returns (bool checkOut)
    {
        Proposal storage prop = proposals[proposalNumber];
        return prop.hashOfProposal == sha3(beneficiaryAddress,
amountOfWei, bytecodeOfTransaction);
```

```
    }


    /**
     * Log some vote for proposal
     *
     * Vote `inSupportOfProposal? to support : against` proposal
#`proposalNumber`
     *
     * @param proposalNumber the number of proposal
     * @param supportsProposalin favor or against it
     */
    function vote(
        uint proposalNumber,
        bool inSupportOfProposal
    )
        onlyShareholders
        returns (uint voteID)
    {
        Proposal storage prop = proposals[proposalNumber];
        require(prop.voted[msg.sender] != true);


        voteID = prop.votes.length++;
        prop.votes[voteID] = Vote({isSupporting:
inSupportOfProposal, voterAddress: msg.sender});
        prop.voted[msg.sender] = true;
        prop.totalVotes = voteID +1;
```

```
        ReadilyVoted(proposalNumber,  inSupportOfProposal,
msg.sender);

        return voteID;

    }



    /**

     * Finish the vote

     *

     * Count votes proposal #`proposalNumber` then execute it if
it is approved

     *

     * @param proposalNumber the proposal number

     * @param bytecodeOfTransaction optional: if the transaction
had a bytecode, then you should send it

     */

    function runProposal(uint proposalNumber, bytes
bytecodeOfTransaction) {

        Proposal storage prop = proposals[proposalNumber];



        require(now > prop.deadlineofVoting
                      // If it’s past voting deadlineForCampaign

            && !prop.executed
                     // and it hasn’t been executed

            && prop.hashOfProposal == sha3(prop.recipient,
prop.coinsAmount, bytecodeOfTransaction)); // and supplied code
matches proposal...




        // ...tally the results
```

```
        uint quorum = 0;
        uint okay = 0;
        uint no = 0;

        for (uint x = 0; x <  prop.votes.length; ++x) {
            Vote storage vs = prop.votes[x];
            uint weightOfVote =
tokenAddressForShares.balanceAmount(vs.voterAddress);
            quorum += weightOfVote;
            if (vs.isSupporting) {
                okay += weightOfVote;
            } else {
                no += weightOfVote;
            }
        }

        require(quorum >= lowestQuorum); // Check whether minimum
quorum has been attained

        if (okay > no ) {
            // Proposal has been passed; execute the transaction

            prop.executed = true;
            require(prop.recipient.call.value(prop.coinsAmount)
(bytecodeOfTransaction));

            prop.passedProposal = true;
```

```
        } else {
            // Proposal has failed
            prop.passedProposal = false;
        }

        // Fire the Events
        TalliedProposal(proposalNumber, okay - no, quorum,
prop.passedProposal);
    }
}
```

Deployment and Usage

You should add the “shares token address” to represent the address of the token to work as a share with the voting rights. It will good for you to note the lines of code described below:

We began by describing the token contract to the new contract. Since it is only using the balanceAmount function, we should only add the single line:

contract myToken { mapping (address => uint256) public balanceAmount; }

We can then define a variable of token type, and it will inherit all the functions that we had described earlier. The token variable can then be pointed to some address on the blockchain, and it can use this to request for live information. This forms the simplest way on how we can make one contract understand one on Ethereum:

```
contract ShareholderAssociation {
    myToken public tokenAddressForShares;
// ...
```

```
function ShareholderAssociation(myToken sharesAddress, uint
minimumSharesForVoting, uint minutesForDebate) {

        tokenAddressForShares = myToken(sharesAddress);
```

This association brings in a new challenge that was not there in the previous Congress. Anyone who has tokens can vote and the balances can change very quickly, it is impossible to count the actual score of a proposal after a shareholder has voted, otherwise, it will be possible for a shareholder to vote multiple times by sending his share to the different addresses. In our contract, we only record the vote position then the real score will be tallied in the execute proposal phase:

```
uint quorum = 0;

uint okay = 0;

uint no = 0;


for (uint i = 0; i <  prop.votes.length; ++i) {

    Vote vs = prop.votes[i];

    uint weightOfVote =
tokenAddressForShares.balanceAmount(vs.voterAddress);

    quorum += weightOfVote;

    if (vs.isSupporting) {

        okay += weightOfVote;

    } else {

        no += weightOfVote;

    }

}
```

Liquid Democracy

Voting on all actions and expenses of a contract requires time and the users are expected to be active constantly, attentive and informed. We can elect an appointed account with control over the contract and we will be able to take swift decisions over it.

We will be implementing a version for liquid democracy, which is simply a highly flexible delegative democracy. In this type of democracy, any voter can become a potential delegate, instead of having to vote the candidate that you need, you only have to say the voter that you trust, and he will handle the decision for you. Your voting weight will then be delegated to them and they can delegate it to another voter whom they trust and so on. The result will be that the account that is voted most will be the one with trust connections to largest coinsAmount of voters:

```
pragma solidity ^0.4.16;


contract myToken {

    mapping (address => uint256) public balanceAmount;

}


contract BlockchainDemocracy {

    myToken public tokenForVoting;

    bool  beingExecuted;

    address public  appointeeAddress;

    mapping (address => uint) public voterId;

    mapping (address => uint256) public weightOfVote;


    uint public percentDelegated;

    uint public calculateLastWeight;

    uint public delegationRounds;
```

```
    uint public totalVotes;

    VoteDelegated[] public delegatedVotes;

    string public functionForbidden;


    event ANewAppointee(address newAppointeeAddress, bool
changed);


    struct VoteDelegated {

        address nomineeAddress;

        address voterAddress;

    }


    /**

     * The constructor function

     */

    function BlockchainDemocracy(

        address votingWeightToken,

        string function ForbiddenCall,

        uint percentLossInEachRound

    ) {

        tokenForVoting = myToken(votingWeightToken);

        delegatedVotes.length++;

        delegatedVotes[0] = VoteDelegated({nomineeAddress: 0,
voterAddress: 0});

        functionForbidden = function ForbiddenCall;
```

```
        percentDelegated = 100 - percentLossInEachRound;
        if (percentDelegated > 100) percentDelegated = 100;
    }

    /**
     * Vote for address
     *
     * Send the vote weight to some other address
     *
     * @param nominatedAddress destination address to receive the
sender's vote
     */
    function vote(address nominatedAddress) returns (uint
voteIndex) {
        if (voterId[msg.sender]== 0) {
            voterId[msg.sender] = delegatedVotes.length;
            totalVotes++;
            voteIndex = delegatedVotes.length++;
            totalVotes = voteIndex;
        }
        else {
            voteIndex = voterId[msg.sender];
        }

        delegatedVotes[voteIndex] =
VoteDelegated({nomineeAddress: nominatedAddress, voterAddress:
msg.sender});
```

```
        return voteIndex;
    }

    /**
     * Perform the Executive Action
     *
     * @param target destination address to interact with
     * @param valueInWei coinsAmount of ether to be send together with the transaction
     * @param bytecode data bytecode for transaction
     */
    function execute(address target, uint valueInWei, bytes32 bytecode) {
        require(msg.sender ==  appointeeAddress // If the caller is our current  appointeeAddress,
            && !beingExecuted // // in case the call is executed,
            && bytes4(bytecode) != bytes4(sha3(functionForbidden))  // and it is not trying to do a forbidden function
            && delegationRounds >= 4);      // and the delegation has been calculated enough

        beingExecuted = true;
        assert(target.call.value(valueInWei)(bytecode)); // Execute the command.
        beingExecuted = false;
    }
```

```
    /**
     * Calculate theVotes
     *
     * Go through all delegated vote logs then tally up every
address's total rank
     */
    function getVotes() returns (address winner) {
        address currentWinnerAddress =  appointeeAddress;
        uint maxCurrently = 0;
        uint weight = 0;
        VoteDelegated storage vs = delegatedVotes[0];

        if (now > calculateLastWeight + 90 minutes) {
            delegationRounds = 0;
            calculateLastWeight = now;

            // Distribute initial weight
            for (uint x=1; x< delegatedVotes.length; x++) {
                weightOfVote[delegatedVotes[x].nomineeAddress] =
0;
            }
            for (x=1; x< delegatedVotes.length; x++) {
                weightOfVote[delegatedVotes[x].voterAddress] =
tokenForVoting.balanceAmount(delegatedVotes[x].voterAddress);
            }
        }
        else {
```

```
            delegationRounds++;

            uint lossRatio = 100 * percentDelegated **
delegationRounds / 100 ** delegationRounds;

            if (lossRatio > 0) {

                for (i=1; i< delegatedVotes.length; i++){

                    vs = delegatedVotes[i];



                    if (vs.nomineeAddress != vs.voterAddress &&
weightOfVote[vs.voterAddress] > 0) {

                        weight = weightOfVote[vs.voterAddress] *
lossRatio / 100;

                        weightOfVote[vs.voterAddress] -= weight;

                        weightOfVote[vs.nomineeAddress] +=
weight;

                    }



                    if (delegationRounds>3 &&
weightOfVote[vs.nomineeAddress] > maxCurrently) {

                        currentWinnerAddress = vs.nomineeAddress;

                        maxCurrently =
weightOfVote[vs.nomineeAddress];

                    }

                }

            }

        }



        if (delegationRounds > 3) {

            ANewAppointee(currentWinnerAddress,  appointeeAddress
== currentWinnerAddress);
```

```
                appointeeAddress = currentWinnerAddress;
            }


            return currentWinnerAddress;
        }
}
```

For you to deploy this, you will need a token. You can use the token you had created previously or just deploy a new one then distribute it amongst the accounts. Next, copy the token address.

Deploy your democracy contract and put the token address on “Voting weight token”, set “Percentage loss in each round” as 75 and the “forbidden function” as transferOwnership(address)without adding any additional characters.

Choosing a Delegate

After deploying the liquid democracy contract, go to its page. Allow any of the shareholders to vote on whom they will trust to make voting decisions for the contract. If you need to be the final decision maker, feel free to vote for yourself.

Once several individuals have voted, execute the “Count Votes” function to calculate the voting weight of everyone. You should execute the function several times, and in the first run, the weight for everyone will be set to their balance in the selected token. The voting weight in the next round will go to the individual whom you voted, in the next run, the voting weight will go to the person voted by the person you voted, and the chain will continue. The vote delegations can lead to infinite loops, but these are prevented as every time that a vote has been voted, it will lose so0me infinite power which is set at “percentLossInEachRound” during contract launch. If you set the loss at 75%, this means that the person you vote will get 100% of your weight, but

they will only forward 75% of their weight in case they delegate the vote to someone else. The person is also allowed to delegate the vote to someone else, but they will only get 56% of your voting weight.

After the expiry of one and half hours from the time the time of calling the "Calculate votes" begins, all the weights will be reset, and they will be recalculated based on the original token balance. This means that once you have received more tokens, it will be good for you to re-execute the function.

The question is, what is the purpose of voter delegation? You can use it rather than the token weight on an Association. Any time that you are defining your contract, it is possible for you to describe several other contracts that will be used by the main contract.

Time-Locked Multisig

This feature can be used as a security measure. We will be working on a code that is based on Congress DAO, but we will twist it. Every transaction will not require the approval by X number of members, but a single member will be able to initiate a transaction, but a minimum coinsAmount of delay will be required before one can execute a transaction, and this will vary based on the support a transaction has. A proposal with a high number of approvals will be executed faster. A member can vote against a transaction, meaning that one of the approved signatures will be canceled.

With no urgency, you will only need one or two signatures to execute any transaction. However, in any case, a single key is compromised, the other keys will delay the transaction for months or a year or even stop the transaction from execution.

Once a transaction has been approved by all the keys, then it can be executed after 10 minutes, but this time can be re-configured. The coinsAmount of time that the transaction requires will double every time once 5% of members don't vote. If they vote against it, this will quadruple. For a simple ether transaction, it will be executed immediately after a vote of support puts the

transaction under required time. A complex transaction will need it to be executed manually with the right bytecode. These are the default values, but one can set them to other values when creating the contract:

- 100% approval: 10 minutes (the minimum default)
- 90% approval: 40 minutes
- 80%: 2h40
- 50%: about a week
- 40%: 1 month
- 30%: 4 months
- 20%: Over a year
- 10% or less: 5 years or never

After the expiry of the minimum coinsAmount of time that is required, anyone will be able to execute the transaction. Here is the code:

```
pragma solidity ^0.4.16;


contract owned {
    address public ownerAddress;


    function owned() {
        ownerAddress = msg.sender;
    }


    modifier  onlyOwnerModifier {
        require(msg.sender == ownerAddress);
```

```
        _;
    }

    function transferOwnership(address newOwner)
 onlyOwnerModifier {
        ownerAddress = newOwner;
    }
}

contract tokenRecipient {
    event etherReceivedEvent(address sender, uint coinsAmount);
    event tokensReceivedEvent(address _from, uint256 _value,
address _token, bytes _extraData);

    function approveReceipt(address _from, uint256 _value,
address _token, bytes _extraData){
        Token tok = Token(_token);
        require(tok.transferSource(_from, this, _value));
        tokensReceivedEvent(_from, _value, _token, _extraData);
    }

    function () payable {
        etherReceivedEvent(msg.sender, msg.value);
    }
}

interface Token {
```

```
    function transferSource(address _from, address _to, uint256
_value) returns (bool successful);

}


contract TimeLock is owned, tokenRecipient {


    Proposal[] public proposals;

    uint public numberOfProposals;

    mapping (address => uint) public idOfMember;

    Member[] public availableMembers;

    uint minimumTime = 10;


    event ProposalCreated(uint idOfProposal, address recipient,
uint coinsAmount, string description);

    event ReadilyVoted(uint idOfProposal, bool position, address
voterAddress, string justificationString);

    event ProposalExecuted(uint idOfProposal, int result, uint
deadlineForCampaign);

    event ChangedMembership(address memberAddress, bool
isMemberBool);


    struct Proposal {

        address recipient;

        uint coinsAmount;

        string description;

        bool executed;

        int obtainedResult;

        bytes32 hashOfProposal;
```

```
        uint creationDate;
        Vote[] votes;
        mapping (address => bool) voted;
    }

    struct Member {
        address memberAddress;
        string name;
        uint memberFrom;
    }

    struct Vote {
        bool isSupporting;
        address voterAddress;
        string justificationString;
    }

    // Modifier to allow only shareholders to create new
proposals and vote
    modifier onlyMembersModifier {
        require(idOfMember[msg.sender] != 0);
        _;
    }

    /**
     * The constructor function
```

```
     *
     * First time setup
     */
    function TimeLock(address founder, address[] initialMembers,
uint minimumAmountOfMinutes) payable {
        if (founder != 0) ownerAddress = founder;
        if (minimumAmountOfMinutes !=0) minimumTime =
minimumAmountOfMinutes;
        // Adding an empty first member
         createMember(0, '');
        // adding the founde, to save a step later
         createMember(ownerAddress, 'founder');
        changeMembers(initialMembers, true);
    }

    /**
     * Adding a member
     *
     * @param targetMemberAddress the address to be added as a
member
     * @param memberName the label to give the member address
     */
    function  createMember(address targetMemberAddress, string
memberName)  onlyOwnerModifier {
        uint id;
        if (idOfMember[targetMemberAddress] == 0) {
            idOfMember[targetMemberAddress] =
availableMembers.length;
```

```
            id = availableMembers.length++;
        } else {
            id = idOfMember[targetMemberAddress];
        }

        availableMembers[id] = Member({memberAddress:
targetMemberAddress, memberFrom: now, name: memberName});
        ChangedMembership(targetMemberAddress, true);
    }

    /**
     * Remove a member
     *
     * @param targetMemberAddress member to be removed
     */
    function deleteMember(address targetMemberAddress)
 onlyOwnerModifier {
        require(idOfMember[targetMemberAddress] != 0);

        for (uint i = idOfMember[targetMemberAddress];
i<availableMembers.length-1; i++){
            availableMembers[i] = availableMembers[i+1];
        }
        delete availableMembers[availableMembers.length-1];
        availableMembers.length--;
    }
```

```
    /**
     * Edit the existing members
     *
     * @param newMembers an array of the addresses to be updated
     * @param isQualifiedToVote a new voting value that all
values are to be set to
     */
    function changeMembers(address[] newMembers, bool
isQualifiedToVote) {
        for (uint x = 0; x < newMembers.length; x++) {
            if (isQualifiedToVote)
                createMember(newMembers[x], '');
            else
                deleteMember(newMembers[x]);
        }
    }


    /**
     * Add a Proposal
     *
     * Propose to send the `amountOfWei / 1e18` ether to a
`beneficiaryAddress` for `jobDescription`. `bytecodeOfTransaction
? Contains : Does not contain` code.
     *
     * @param beneficiaryAddress who to send the ether to
     * @param amountOfWei coinsAmount of ether to send, in wei
     * @param jobDescription a description of the job
```

```
     * @param bytecodeOfTransaction the bytecode of the
transaction

     */

    function createNewProposal(

        address beneficiaryAddress,

        uint amountOfWei,

        string jobDescription,

        bytes bytecodeOfTransaction

    )

        onlyMembersModifier

        returns (uint idOfProposal)

    {

        idOfProposal = proposals.length++;

        Proposal storage prop = proposals[idOfProposal];

        prop.recipient = beneficiaryAddress;

        prop.coinsAmount = amountOfWei;

        prop.description = jobDescription;

        prop.hashOfProposal = sha3(beneficiaryAddress,
amountOfWei, bytecodeOfTransaction);

        prop.executed = false;

        prop.creationDate = now;

        ProposalCreated(idOfProposal, beneficiaryAddress,
amountOfWei, jobDescription);

        numberOfProposals = idOfProposal+1;

        vote(idOfProposal, true, '');



        return idOfProposal;
```

```
    }


    /**
     * Add a proposal in Ether
     *
     * Propose to send `etherAmount` ether to
`beneficiaryAddress` for `jobDescription`. `bytecodeOfTransaction
? Contains : Does not contain` code.
     * This is a convenience function to use if the coinsAmount
to be given is in round number of ether units.
     *
     * @param beneficiaryAddress whom to send ether to
     * @param etherAmount the coinsAmount of ether to be send
     * @param jobDescription a description of a job
     * @param bytecodeOfTransaction the bytecode of transaction
     */
    function newProposal(
        address beneficiaryAddress,
        uint etherAmount,
        string jobDescription,
        bytes bytecodeOfTransaction
    )
        onlyMembersModifier
        returns (uint idOfProposal)
    {
        return createNewProposal(beneficiaryAddress, etherAmount
* 1 ether, jobDescription, bytecodeOfTransaction);
```

```
    }

    /**
     * Check whether the proposal code matches
     *
     * @param proposalNumber the ID number of proposal to query
     * @param beneficiaryAddress whom to send ether to
     * @param amountOfWei the coinsAmount of ether to be send
     * @param bytecodeOfTransaction the bytecode of the
transaction
     */
    function checkCode(
        uint proposalNumber,
        address beneficiaryAddress,
        uint amountOfWei,
        bytes bytecodeOfTransaction
    )
        constant
        returns (bool checkOut)
    {
        Proposal storage prop = proposals[proposalNumber];
        return prop.hashOfProposal == sha3(beneficiaryAddress,
amountOfWei, bytecodeOfTransaction);
    }

    /**
```

```
     * Log a vote for the proposal
     *
     * Vote `inSupportOfProposal? in support of : against`
proposal #`proposalNumber`
     *
     * @param proposalNumber the number of proposal
     * @param inSupportOfProposal in favor or against it
     * @param justificationString an optional justificationString
text
     */
    function vote(
        uint proposalNumber,
        bool inSupportOfProposal,
        string justificationString
    )
        onlyMembersModifier
    {
        Proposal storage prop = proposals[proposalNumber]; // Get
proposal
        require(prop.voted[msg.sender] != true);           // If
already voted, cancel
        prop.voted[msg.sender] = true;                  // Set the
voter as having voted
        if (inSupportOfProposal) {                      // If he
supports the proposal
            prop.obtainedResult++;                   // Increase
the score
        } else {                                    // If he doesn't
            prop.obtainedResult--;                   // Decrease
```

```
the score

        }


        // Create a log for the event

        ReadilyVoted(proposalNumber,  inSupportOfProposal,
msg.sender, justificationString);


        // Execute it now if possible

        if ( now > proposalDeadline(proposalNumber)

            && prop.obtainedResult > 0

            && prop.hashOfProposal == sha3(prop.recipient,
prop.coinsAmount, '')

            && inSupportOfProposal) {

            runProposal(proposalNumber, '');

        }

    }


    function proposalDeadline(uint proposalNumber) constant
returns(uint deadlineForCampaign) {

        Proposal storage prop = proposals[proposalNumber];

        uint factor = calculateFactor(uint(prop.obtainedResult),
(availableMembers.length - 1));

        return prop.creationDate + uint(factor * minimumTime *  1
minutes);

    }


    function calculateFactor(uint a, uint b) constant returns
(uint factor) {
```

```
        return 2**(20 - (20 * a)/b);

    }


    /**

     * Finish the vote

     *

     * Count votes proposal #`proposalNumber` then execute it if
it is approved

     *

     * @param proposalNumber the proposal number

     * @param bytecodeOfTransaction optional: if transaction had
a bytecode, you should send it

     */

    function runProposal(uint proposalNumber, bytes
bytecodeOfTransaction) {

        Proposal storage prop = proposals[proposalNumber];



        require(now >= proposalDeadline(proposalNumber)
                            // If it’s past voting deadline

            && prop.obtainedResult > 0
                                // and the minimum quorum has
been reached

            && !prop.executed
                               // and it’s not being executed
currently

            && checkCode(proposalNumber, prop.recipient,
prop.coinsAmount, bytecodeOfTransaction));  // and supplied code
matches proposal...
```

```
        prop.executed = true;

        assert(prop.recipient.call.value(prop.coinsAmount)
(bytecodeOfTransaction));



        // Fire the Events

        ProposalExecuted(proposalNumber, prop.obtainedResult,
proposalDeadline(proposalNumber));

    }

}
```

The deployment of the code should be done in the same way you have been doing before. For the case of the deployment parameters, if you leave the field for minimum time blank, it will default to 30 minutes. If you need to have faster lock times, you can set it to 1 minute. Once you have uploaded it, run the “Add Members” function to add new members of the group. These can be people you know or accounts on offline computers or on different computers.

The account that is set as “Owner” is very powerful since it can be used to add or remove members at will. Once you have added the main members, it will be good for you to set the “Owner” to some other account. To do this, you only must execute the “Transfer Membership” function. This can be set to multisig itself if you need all the removals or additions of members to be voted, similarly to any other transaction. You can also choose to set this to another trusted multisig wallet or set it to 0x000 if you want to remain with a fixed number of members forever. It is good for you to note that the funds held in the account will only be safe as the “owner” account.

For the sake of simplicity, a vote against a particular proposal only counts as a single vote less the vote support. It is good for you to play around with the idea that the negative votes weight more, but this is an indication that minority of the members could possess and effective veto power on any transaction that is proposed.

Chapter 18

Creating a Token

In this chapter, we will be creating a digital token. In Ethereum, tokens can be used to represent any tradable goods such as gold certificates, loyalty certificates, coins, in Game items, IOUs, and others. Tokens are used to implement basic features in a standard way, making it compatible with any Ethereum components such as wallets that are developed based on such standards. Below is an example of a basic token in Ethereum:

```
contract MyToken {

    /* This will create an array with all the balances */

    mapping (address => uint256) public balanceAmount;


    /* Initializes the contract with the initial supply tokens to
creator of the contract */

    function MyToken(

        uint256 initialAmount

        ) {

        balanceAmount[msg.sender] = initialAmount;    // Giving
the creator all the initial tokens

    }


    /* Sending the coins */
```

```
    function transfer(address _to, uint256 _value) {

        require(balanceAmount[msg.sender] >= _value); // Checking
if the sender is having enough

        require(balanceAmount[_to] + _value >=
balanceAmount[_to]); // Checking for the overflows

        balanceAmount[msg.sender] -= _value;     // Subtracting
from the sender

        balanceAmount[_to] += _value;         // Adding a similar
coinsAmount to the recipient

    }

}
```

The following is the full code that can help you create a token:

pragma solidity ^0.4.16;

interface tokenRecipient { function approveReceipt(address _from, uint256 _value, address _token, bytes _extraData) public; }

```
contract ERC20Token {

    // Public variables for the token

    string public name;

    string public tokenSymbol;

    uint8 public decimals = 18;

    // 18 decimals is the strongly suggested default, avoid
changing it

    uint256 public supplied;


    // Creating an array with all the balances
```

```
    mapping (address => uint256) public balanceAmount;

    mapping (address => mapping (address => uint256)) public
allowance;



    // Generating a public event on our blockchain to notify the
clients

    event Transfer(address indexed from, address indexed to,
uint256 value);



    // To notify the clients about coinsAmount burnt

    event BurnEvent(address indexed from, uint256 value);



    /**

     * The constructor function

     *

     * Initializing the contract with the initial supply tokens
to creator of contract

     */

    function ERC20Token(

        uint256 initialAmount,

        string tokenName,

        string tokenSymbol

    ) public {

        supplied = initialAmount * 10 ** uint256(decimals); //
Update the total supply with decimal coinsAmount

        balanceAmount[msg.sender] = supplied;// Give the creator
all the initial tokens

        name = tokenName;    // Set name for the display purposes
```

```
        tokenSymbol = tokenSymbol; // Set symbol for purposes of
display

    }



    /**

     * Internal transfer, it can only be called by this contract

     */

    function _transfer(address _from, address _to, uint _value)
internal {

        // Preventing a transfer to address 0x0. Use burn()
instead

        require(_to != 0x0);

        // Checking whether the sender has enough

        require(balanceAmount[_from] >= _value);

        // Checking the overflows

        require(balanceAmount[_to] + _value >
balanceAmount[_to]);

        // Saving it for an assertion in future

        uint prevBalances = balanceAmount[_from] +
balanceAmount[_to];

        // Subtracting from the sender

        balanceAmount[_from] -= _value;

        // Adding same to recipient

        balanceAmount[_to] += _value;

        Transfer(_from, _to, _value);

        // Asserts are normally used to use a static analysis for
finding bugs in the code. They shouldn't fail

        assert(balanceAmount[_from] + balanceAmount[_to] ==
prevBalances);
```

```
    }

    /**
     * Transferring the tokens
     *
     * Send `_value` tokens to `_to` from your account
     *
     * @param _to address of the recipient
     * @param _value coinsAmount to send
     */
    function transfer(address _to, uint256 _value) public {
        _transfer(msg.sender, _to, _value);
    }

    /**
     * Transfer tokens from the other address
     *
     * Send the `_value` tokens to the `_to` in behalf of `_from`
     *
     * @param _from address of sender
     * @param _to address of recipient
     * @param _value coinsAmount to send
     */
    function transferSource(address _from, address _to, uint256
_value) public returns (bool successful) {
        require(_value <= allowance[_from][msg.sender]); // Check
```

```
the allowance
        allowance[_from][msg.sender] -= _value;
        _transfer(_from, _to, _value);
        return true;
    }

    /**
     * Set the allowance for the other address
     *
     * Allows the `_spender` to spend less than `_value` tokens
on your behalf
     *
     * @param _spender address authorized to spend
     * @param _value maximum coinsAmount they are allowed to
spend
     */
    function approveFunction(address _spender, uint256 _value)
public
        returns (bool successful) {
        allowance[msg.sender][_spender] = _value;
        return true;
    }

    /**
     * Setting allowance for the other address then notify
     *
     * Allow `_spender` to spend no more than `_value` tokens on
```

```
your behalf, then ping the contract regarding this

     *

     * @param _spender The address authorized to spend

     * @param _value the max coinsAmount they can spend

     * @param _extraData some extra information to send to the
approved contract

     */

    function approveAndCall(address _spender, uint256 _value,
bytes _extraData)

        public

        returns (bool successful) {

        tokenRecipient spender = tokenRecipient(_spender);

        if (approveFunction(_spender, _value)) {

            spender.approveReceipt(msg.sender, _value, this,
_extraData);

            return true;

        }

    }



    /**

     * Destroy the tokens

     *

     * Remove the `_value` tokens from system irreversibly

     *

     * @param _value coinsAmount of money to be burned

     */

    function burn(uint256 _value) public returns (bool
```

```
successful) {

        require(balanceAmount[msg.sender] >= _value); // Check
whether the sender has enough

        balanceAmount[msg.sender] -= _value;   // Subtract from
sender

        supplied -= _value;          // Updates the supplied

        BurnEvent(msg.sender, _value);

        return true;

    }



    /**

     * Destroy the tokens from the other account

     *

     * Remove the `_value` tokens from system irreversibly on the
behalf of `_from`.

     *

     * @param _from address of sender

     * @param _value coinsAmount of money to be burned

     */

    function burnSource(address _from, uint256 _value) public
returns (bool successful) {

        require(balanceAmount[_from] >= _value); // Checking
whether the targeted balance is adequate

        require(_value <= allowance[_from][msg.sender]); //
Checking the allowance

        balanceAmount[_from] -= _value; // Subtract from targeted
balance

        allowance[_from][msg.sender] -= _value; // Subtract from
sender's allowance
```

```
        supplied -= _value;  // Update the supplied
        BurnEvent(_from, _value);
        return true;
    }
}
```

Open the Wallet app then go to the “Contracts” tab then “Deploy New Contract”. In the text field for “Solidity Contract Source code”, just type the code given below:

```
    contract MyToken {
        /* This will create an array with all the balances */
        mapping (address => uint256) public balanceAmount;
    }
```

A mapping is simply an association array, in which addresses are associated with balances. The addresses are normally written in the hexadecimal format, while the balances are in the form of integers, which range between 0 and 115 quattuorvigintillion. A quattuorvigintillion is many vigintillions more than what you are planning to use your tokens for. Note that we have used the “public” keyword, which means that anyone on the blockchain will be able to access the variable, which means that all the balances will be public. This is how they should be for the clients to display them.

If you just deployed your contract straight away, it is working but not much useful. You can use the contract to query the balances of your coin on various addresses, but since you have not created a single coin for any of them, they will return a value of 0.

We will be creating some few tokens on startup. Add the following code before the last closing bracket and under “mapping…” line:

```
    function MyToken() {
        balanceAmount[msg.sender] = 21000000;
```

```
}
```

Note that we used the same name for the function “MyToken” as the contract “MyToken”. The two must match, so if you rename one, ensure that you rename the other one. It is a startup, special function that will run only once when the contract is uploaded to the network for the first time. The function is expected to set the balance who deploys the contract, that is, msg.sender, to 21 million. Note that you don’t have to stick to 21 million, but you can choose any value that you want. However, instead of doing it, as shown above, you can choose to supply it as a parameter to the function as demonstrated below:

```
function MyToken(uint256 initialAmount) {

    balanceAmount[msg.sender] = initialAmount;

}
```

You can look at the right column and look for a drop down which will be written “pick a contract”. Choose “MyToken” contract and you will see it display a section for “Constructor parameters”. These are parameters for the token which can be changed, meaning that the piece of code can be reused by changing those parameters.

At this point, we have a functional contract that has created balances of tokens but there is no function to move it, so it only stays on the same account. This is what we need to implement now. Just add the code given below before our last bracket:

```
/* Send the coins */

function transfer(address _to, uint256 _value) {

    /* Add then subtract the new balances */

    balanceAmount[msg.sender] -= _value;

    balanceAmount[_to] += _value;

}
```

The function takes two parameters, the recipient, and a value. Once the

function is called, it will subtract _value from balance then add it to the _to balance. A problem will arise when one needs to send more than what he owns. We need to stop the contract from executing in case of such an occurrence. This can be achieved by use of a “throw” or “return” as shown below:

```
    function transfer(address _to, uint256 _value) {

        /* Check whether the sender has a balance and for
overflows */

        require(balanceAmount[msg.sender] >= _value &&
balanceAmount[_to] + _value >= balanceAmount[_to]);



        /* Add the subtract the new balances */

        balanceAmount[msg.sender] -= _value;

        balanceAmount[_to] += _value;

    }
```

We now miss some basic information about the contract. This can be used using a token registry, but let us add them directly:

```
string public name;

string public tokenSymbol;

uint8 public decimals;
```

We can then update the constructor function so that the variables can be set at the start:

```
/* Initialize the contract with the initial supply of tokens to
creator of the contract */

    function MyToken(uint256 initialAmount, string tokenName,
string tokenSymbol, uint8 decimalUnits) {

        balanceAmount[msg.sender] = initialAmount;   // Give the
creator all the initial tokens

        name = tokenName;       // Set a name for purposes of
```

```
display

        tokenSymbol = tokenSymbol;   // Set a symbol for purposes
of display         decimals = decimalUnits;         // Amount of
decimals for purposes of display

    }
```

Finally, we need Events. These are empty and special functions that one can call to help clients such as Ethereum wallet to track the activities that happen in the contract. An event name should begin with a capital letter. Add the following to the beginning of the contract to define an event:

```
event Transfer(address indexed from, address indexed to, uint256
value);
```

```
You can then add the following lines to your transfer function:

        /* Notify anyone who is listening that the transfer took
place */

        Transfer(msg.sender, _to, _value);
```

At this point, the token is ready.

Deploying the Contract

It is now time for us to deploy the contract. Open Ethereum wallet, move to contracts tab then click “deploy new contract”. Paste your code into “Solidity source field”. If your code has no error, you will see the drop down for “pick a contract” on the right. Choose “MyToken” contract. All the parameters that you should personalize for your token will be shown on the right. We recommend you set them as follows: 10,000 as supply, type any name you want, % for symbol then 2 decimal places.

Scroll to the bottom of the page and see the details for the execution of your contract. Press “deploy” then type the password for your account. In the next page, click the account with the name “Etherbase”.

Conclusion

This marks the end of this book. At this point, you should be familiar with nearly all concepts involved in the development of decentralized as discussed in this book. The concept of decentralized applications is one of the novel ideas that have emerged from the blockchain technology. The fact that decentralized applications can run without the control of a centralized entity, board or any single individual has seen many organizations express their interest in them.

One last thing. How would you like winning a **$200.00 Amazon Gift card** and helping us improve this book in the process with a little bit of **feedback**?

That's right :)

Your opinion is so valuable to us that we are giving away a $200 gift card to the luckiest *one of 200 participants*!

It will only take *a minute of your time* to let us know what you like and what you didn't like about this book. The hardest part is deciding how to spend the two hundred dollars!

Just follow this link.

http://reviewers.win/dapps