Introduction to Smart Contracts

A Simple Smart Contract

Let us begin with the most basic example. It is fine if you do not understand everything right now, we will go into more detail later.

Storage

```
pragma solidity ^0.4.0;

contract SimpleStorage {
    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public view returns (uint) {
        return storedData;
    }
}
```

The first line simply tells that the source code is written for Solidity version 0.4.0 or anything newer that does not break functionality (up to, but not including, version 0.5.0). This is to ensure that the contract does not suddenly behave differently with a new compiler version. The keyword `pragma` is called that because, in general, pragmas are instructions for the compiler about how to treat the source code (e.g. pragma once).

A contract in the sense of Solidity is a collection of code (its *functions*) and data (its *state*) that resides at a specific address on the Ethereum blockchain. The line `uint storedData;` declares a state variable called `storedData` of type `uint` (unsigned integer of 256 bits). You can think of it as a single slot in a database that can be queried and altered by calling functions of the code that manages the database. In the case of Ethereum, this is always the owning contract. And in this case, the functions `set` and `get` can be used to modify or retrieve the value of the variable.

To access a state variable, you do not need the prefix `this.` as is common in other languages.

This contract does not do much yet (due to the infrastructure built by Ethereum) apart from allowing anyone to store a single number that is accessible by anyone in the world without a (feasible) way to prevent you from publishing this number. Of course, anyone could just call `set` again with a different value and overwrite your number, but the number will still be stored in the history of the blockchain. Later, we will see how you can impose access restrictions so that only you can alter the number.

Note

All identifiers (contract names, function names and variable names) are restricted to the ASCII character set. It is possible to store UTF-8 encoded data in string variables.

Warning

Be careful with using Unicode text, as similar looking (or even identical) characters can have different code points and as such will be encoded as a different byte array.

Subcurrency Example

The following contract will implement the simplest form of a cryptocurrency. It is possible to generate coins out of thin air, but only the person that created the contract will be able to do that (it is trivial to implement a different issuance scheme). Furthermore, anyone can send coins to each other without any need for registering with username and password — all you need is an Ethereum keypair.

```
pragma solidity >0.4.24;

contract Coin {
    // The keyword "public" makes those variables
    // readable from outside.
    address public minter;
    mapping (address => uint) public balances;

    // Events allow light clients to react to
    // changes efficiently.
    event Sent(address from, address to, uint amount);

    // This is the constructor whose code is
    // run only when the contract is created.
    constructor() public {
        minter = msg.sender;
    }

    function mint(address receiver, uint amount) public {
```

```
        if (msg.sender != minter) return;
        balances[receiver] += amount;
    }

    function send(address receiver, uint amount) public {
        if (balances[msg.sender] < amount) return;
        balances[msg.sender] -= amount;
        balances[receiver] += amount;
        emit Sent(msg.sender, receiver, amount);
    }
}
```

This contract introduces some new concepts, let us go through them one by one.

The line `address public minter;` declares a state variable of type address that is publicly accessible. The `address` type is a 160-bit value that does not allow any arithmetic operations. It is suitable for storing addresses of contracts or keypairs belonging to external persons. The keyword `public` automatically generates a function that allows you to access the current value of the state variable from outside of the contract. Without this keyword, other contracts have no way to access the variable. The code of the function generated by the compiler is roughly equivalent to the following:

```
function minter() returns (address) { return minter; }
```

Of course, adding a function exactly like that will not work because we would have a function and a state variable with the same name, but hopefully, you get the idea - the compiler figures that out for you.

The next line, `mapping (address => uint) public balances;` also creates a public state variable, but it is a more complex datatype. The type maps addresses to unsigned integers. Mappings can be seen as hash tables which are virtually initialized such that every possible key exists and is mapped to a value whose byte-representation is all zeros. This analogy does not go too far, though, as it is neither possible to obtain a list of all keys of a mapping, nor a list of all values. So either keep in mind (or better, keep a list or use a more advanced data type) what you added to the mapping or use it in a context where this is not needed, like this one. The getter function created by the `public` keyword is a bit more complex in this case. It roughly looks like the following:

```
function balances(address _account) public view returns (uint) {
    return balances[_account];
}
```

As you see, you can use this function to easily query the balance of a single account.

The line `event Sent(address from, address to, uint amount);` declares a so-called "event" which is emitted in the last line of the function `send`. User interfaces (as well as server applications of course) can listen for those events being emitted on the blockchain without much cost. As soon as it is emitted, the listener will also receive the arguments `from`, `to` and `amount`, which makes it easy to track transactions. In order to listen for this event, you would use

```
Coin.Sent().watch({}, '', function(error, result) {
    if (!error) {
        console.log("Coin transfer: " + result.args.amount +
            " coins were sent from " + result.args.from +
            " to " + result.args.to + ".");
        console.log("Balances now:\n" +
            "Sender: " + Coin.balances.call(result.args.from) +
            "Receiver: " + Coin.balances.call(result.args.to));
    }
})
```

Note how the automatically generated function `balances` is called from the user interface.

The special function `Coin` is the constructor which is run during creation of the contract and cannot be called afterwards. It permanently stores the address of the person creating the contract: `msg` (together with `tx` and `block`) is a magic global variable that contains some properties which allow access to the blockchain. `msg.sender` is always the address where the current (external) function call came from.

Finally, the functions that will actually end up with the contract and can be called by users and contracts alike are `mint` and `send`. If `mint` is called by anyone except the account that created the contract, nothing will happen. On the other hand, `send` can be used by anyone (who already has some of these coins) to send coins to anyone else. Note that if you use this contract to send coins to an address, you will not see anything when you look at that address on a blockchain explorer, because the fact that you sent coins and the changed balances are only stored in the data storage of this particular coin contract. By the use of events it is relatively easy to create a "blockchain explorer" that tracks transactions and balances of your new coin.

Blockchain Basics

Blockchains as a concept are not too hard to understand for programmers. The reason is that most of the complications (mining, hashing, elliptic-curve cryptography, peer-to-peer networks, etc.) are just there to provide a certain set of features and promises. Once you accept these features as

given, you do not have to worry about the underlying technology - or do you have to know how Amazon's AWS works internally in order to use it?

Transactions

A blockchain is a globally shared, transactional database. This means that everyone can read entries in the database just by participating in the network. If you want to change something in the database, you have to create a so-called transaction which has to be accepted by all others. The word transaction implies that the change you want to make (assume you want to change two values at the same time) is either not done at all or completely applied. Furthermore, while your transaction is applied to the database, no other transaction can alter it.

As an example, imagine a table that lists the balances of all accounts in an electronic currency. If a transfer from one account to another is requested, the transactional nature of the database ensures that if the amount is subtracted from one account, it is always added to the other account. If due to whatever reason, adding the amount to the target account is not possible, the source account is also not modified.

Furthermore, a transaction is always cryptographically signed by the sender (creator). This makes it straightforward to guard access to specific modifications of the database. In the example of the electronic currency, a simple check ensures that only the person holding the keys to the account can transfer money from it.

Blocks

One major obstacle to overcome is what, in Bitcoin terms, is called a "double-spend attack": What happens if two transactions exist in the network that both want to empty an account, a so-called conflict?

The abstract answer to this is that you do not have to care. An order of the transactions will be selected for you, the transactions will be bundled into what is called a "block" and then they will be executed and distributed among all participating nodes. If two transactions contradict each other, the one that ends up being second will be rejected and not become part of the block.

These blocks form a linear sequence in time and that is where the word "blockchain" derives from. Blocks are added to the chain in rather regular intervals - for Ethereum this is roughly every 17 seconds.

As part of the "order selection mechanism" (which is called "mining") it may happen that blocks are reverted from time to time, but only at the "tip" of the chain. The more blocks that are added on top, the less likely it is. So it might be that your transactions are reverted and even removed from the blockchain, but the longer you wait, the less likely it will be.

Note

Transactions are not guaranteed to happen on the next block or any future specific block, since it is up to the miners to include transactions and not the submitter of the transaction. This applies to function calls and contract creation transactions alike.

If you want to schedule future calls of your contract, you can use the alarm clock service.

The Ethereum Virtual Machine

Overview

The Ethereum Virtual Machine or EVM is the runtime environment for smart contracts in Ethereum. It is not only sandboxed but actually completely isolated, which means that code running inside the EVM has no access to network, filesystem or other processes. Smart contracts even have limited access to other smart contracts.

Accounts

There are two kinds of accounts in Ethereum which share the same address space: **External accounts** that are controlled by public-private key pairs (i.e. humans) and **contract accounts** which are controlled by the code stored together with the account.

The address of an external account is determined from the public key while the address of a contract is determined at the time the contract is created (it is derived from the creator address and the number of transactions sent from that address, the so-called "nonce").

Regardless of whether or not the account stores code, the two types are treated equally by the EVM.

Every account has a persistent key-value store mapping 256-bit words to 256-bit words called **storage**.

Furthermore, every account has a **balance** in Ether (in "Wei" to be exact) which can be modified by sending transactions that include Ether.

Transactions

A transaction is a message that is sent from one account to another account (which might be the same or the special zero-account, see below). It can include binary data (its payload) and Ether.

If the target account contains code, that code is executed and the payload is provided as input data.

If the target account is the zero-account (the account with the address `0`), the transaction creates a **new contract**. As already mentioned, the address of that contract is not the zero address but an address derived from the sender and its number of transactions sent (the "nonce"). The payload of such a contract creation transaction is taken to be EVM bytecode and executed. The output of this execution is permanently stored as the code of the contract. This means that in order to create a contract, you do not send the actual code of the contract, but in fact code that returns that code when executed.

> **Note**
>
> While a contract is being created, its code is still empty. Because of that, you should not call back into the contract under construction until its constructor has finished executing.

Gas

Upon creation, each transaction is charged with a certain amount of **gas**, whose purpose is to limit the amount of work that is needed to execute the transaction and to pay for this execution. While the EVM executes the transaction, the gas is gradually depleted according to specific rules.

The **gas price** is a value set by the creator of the transaction, who has to pay `gas_price * gas` up front from the sending account. If some gas is left after the execution, it is refunded in the same way.

If the gas is used up at any point (i.e. it is negative), an out-of-gas exception is triggered, which reverts all modifications made to the state in the current call frame.

Any unused gas is refunded at the end of the transaction.

Storage, Memory and the Stack

The Ethereum Virtual Machine has three areas where it can store data.

Each account has a data area called **storage**, which is persistent between function calls. Storage is a key-value store that maps 256-bit words to 256-bit words. It is not possible to enumerate storage from within a contract and it is comparatively costly to read, and even more to modify storage. A contract can neither read nor write to any storage apart from its own.

The second data area is called **memory**, of which a contract obtains a freshly cleared instance for each message call. Memory is linear and can be addressed at byte level, but reads are limited to a width of 256 bits, while writes can be either 8 bits or 256 bits wide. Memory is expanded by a word (256-bit), when accessing (either reading or writing) a previously untouched memory word (ie. any offset within a word). At the time of expansion, the cost in gas must be paid. Memory is more costly the larger it grows (it scales quadratically).

The EVM is not a register machine but a stack machine, so all computations are performed on an data area called the **stack**. It has a maximum size of 1024 elements and contains words of 256 bits. Access to the stack is limited to the top end in the following way: It is possible to copy one of the topmost 16 elements to the top of the stack or swap the topmost element with one of the 16 elements below it. All other operations take the topmost two (or one, or more, depending on the operation) elements from the stack and push the result onto the stack. Of course it is possible to move stack elements to storage or memory, but it is not possible to just access arbitrary elements deeper in the stack without first removing the top of the stack.

Instruction Set

The instruction set of the EVM is kept minimal in order to avoid incorrect implementations which could cause consensus problems. All instructions operate on the basic data type, 256-bit words. The usual arithmetic, bit, logical and comparison operations are present. Conditional and unconditional jumps are possible. Furthermore, contracts can access relevant properties of the current block like its number and timestamp.

Message Calls

Contracts can call other contracts or send Ether to non-contract accounts by the means of message calls. Message calls are similar to transactions, in that they have a source, a target, data payload, Ether, gas and return data. In fact, every transaction consists of a top-level message call which in turn can create further message calls.

A contract can decide how much of its remaining **gas** should be sent with the inner message call and how much it wants to retain. If an out-of-gas exception happens in the inner call (or any other exception), this will be signaled by an error value put onto the stack. In this case, only the gas sent together with the call is used up. In Solidity, the calling contract causes a manual exception by

default in such situations, so that exceptions "bubble up" the call stack.

As already said, the called contract (which can be the same as the caller) will receive a freshly cleared instance of memory and has access to the call payload - which will be provided in a separate area called the **calldata**. After it has finished execution, it can return data which will be stored at a location in the caller's memory preallocated by the caller.

Calls are **limited** to a depth of 1024, which means that for more complex operations, loops should be preferred over recursive calls.

Delegatecall / Callcode and Libraries

There exists a special variant of a message call, named **delegatecall** which is identical to a message call apart from the fact that the code at the target address is executed in the context of the calling contract and `msg.sender` and `msg.value` do not change their values.

This means that a contract can dynamically load code from a different address at runtime. Storage, current address and balance still refer to the calling contract, only the code is taken from the called address.

This makes it possible to implement the "library" feature in Solidity: Reusable library code that can be applied to a contract's storage, e.g. in order to implement a complex data structure.

Logs

It is possible to store data in a specially indexed data structure that maps all the way up to the block level. This feature called **logs** is used by Solidity in order to implement **events**. Contracts cannot access log data after it has been created, but they can be efficiently accessed from outside the blockchain. Since some part of the log data is stored in bloom filters, it is possible to search for this data in an efficient and cryptographically secure way, so network peers that do not download the whole blockchain ("light clients") can still find these logs.

Create

Contracts can even create other contracts using a special opcode (i.e. they do not simply call the zero address). The only difference between these **create calls** and normal message calls is that the payload data is executed and the result stored as code and the caller / creator receives the address of the new contract on the stack.

Deactivate and Self-destruct

The only way to remove code from the blockchain is when a contract at that address performs the `selfdestruct` operation. The remaining Ether stored at that address is sent to a designated target and then the storage and code is removed from the state. Removing the contract in theory sounds like a good idea, but it is potentially dangerous, as if someone sends Ether to removed contracts, the Ether is forever lost.

Note

Even if a contract's code does not contain a call to `selfdestruct`, it can still perform that operation using `delegatecall` or `callcode`.

If you want to deactivate your contracts, you should instead **disable** them by changing some internal state which causes all functions to revert. This makes it impossible to use the contract, as it returns Ether immediately.

Warning

Even if a contract is removed by "selfdestruct", it is still part of the history of the blockchain and probably retained by most Ethereum nodes. So using "selfdestruct" is not the same as deleting data from a hard disk.

Previous Next