Ethereum Smart contracts development

With Javascript (2022)

Andrea Lisi, andrealisi.12lj@gmail.com

Part 1
Solidity overview

A brief summary of a Solidity smart contract

Smart contracts: structure

A smart contract is similar to a Java class

It is composed by:

- Declaration
- A State (attributes)
- A list of functions (methods)

```
contract MyContract {
   // State
   uint public value;

   // Functions
   constructor() public {
       value = 1;
   }
   function increase() public {
      value = value+1;
   }
}
```

Smart contracts: state

State variables determine the state of that smart contract

Solidity supports various data types:

- Fixed length
  - *bool, (u)int, bytes32, address*
- Variable length
  - *bytes, string*
- *array, mapping(key_type => value_type)*

Smart contracts: state

Array

- Fixed length or dynamic length, can be iterated over
- Removing an element requires a decision
  - Leaving a blank hole, replacing with last element (breaks ordering), shifting elements (costly)

Mapping(*key* => *value*)

- All non-assigned *values* are Zero (false for bool, 0 for uint, etc)
- Support random access, it is not possible to iterate over the *keys* unless you keep a separate list of all the *keys* with significant value

Smart contracts: functions

Functions compose the code of the smart contract

Functions have labels that declare how they interact with the state:

- A **view** function **only reads** the state;
- A **pure** function does not read or write the state
- Otherwise, the function writes (and reads) the state
    - The state modification will be placed in a transaction
    - It will be written on the blockchain
    - Therefore, it costs a fee to the user

Smart contracts: functions

```
uint public counter;

function increment() public {
    counter = counter + 1;
}

function getSquare() public view returns(uint) {
    return counter**2;
}

function computeSquareOf(uint _a) public pure returns(uint) {
    return _a**2;
}
```

Smart contracts: visibility

State variables and functions can have different visibilities

- **Private**
  - A private state variable or function is exposed only to the contract itself
- **Public**
  - A public function is exposed to other contracts; a public state is a shortcut that creates a getter function with the name of the variable

Smart contracts: visibility

State variables and functions can have different visibilities

- **Internal**
  - An internal state variable or function is exposed to child contracts and the contract itself
- **External**
  - (Only functions) An external function is exposed **only** to other contracts. They are more efficient with large inputs
    - **Warning:** *foo()* does not work; *this.foo()* does
    - https://ethereum.stackexchange.com/questions/19380/external-vs-public-best-practices

Smart contracts: functions

Private does not mean “hidden” or “secret”

- It means a function cannot be called by other smart contracts
    - Only by the contract itself

Remember a Solidity smart contract lives on the Ethereum blockchain, that is visible by anyone

- Can be explored online with explorers
    - Etherscan is one example

Smart contracts: functions

Signature of invoked function

Input sent (1847883)

See? Input is visible (econded in bytes)

https://etherscan.io/tx/0xdbc5b21b0e67731b07dde8fe882975f7d24bd62a76c766d99c414626c189ac4e

Accounts

In Ethereum any entity (account) has associated

- An address: e.g 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
- A balance in Ether greater or equal than 0

The two types of accounts are:

- **Contract Accounts:** are controlled by code, and a transaction activates its code
- **Externally Owned Accounts (EOA):** are controlled by private keys and sign transactions

Global variables

Solidity defines various global variables and functions

- Ether units: *wei*, *gwei*, *szabo*, …
- Time units: *seconds*, *minutes*, …
- Functions: *keccak256*, *abi.encode*, *abi.decode*, …
- Transaction data: *msg*
  - *msg.sender*: the transaction sender (address)
  - *msg.value*: the transaction associated ETH (uint)
- ...

https://docs.soliditylang.org/en/v0.8.3/units-and-global-variables.html

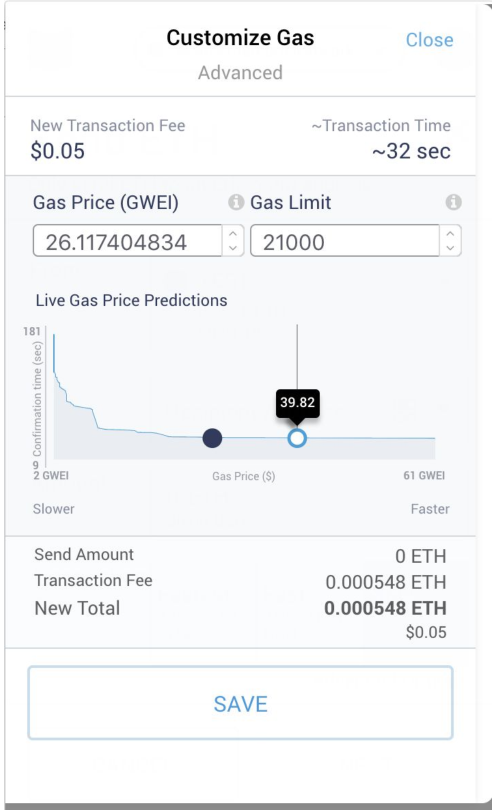
Fees and gas

A function modifying the state writes data on the blockchain

- It requires a transaction

Each transaction costs a fee to the user

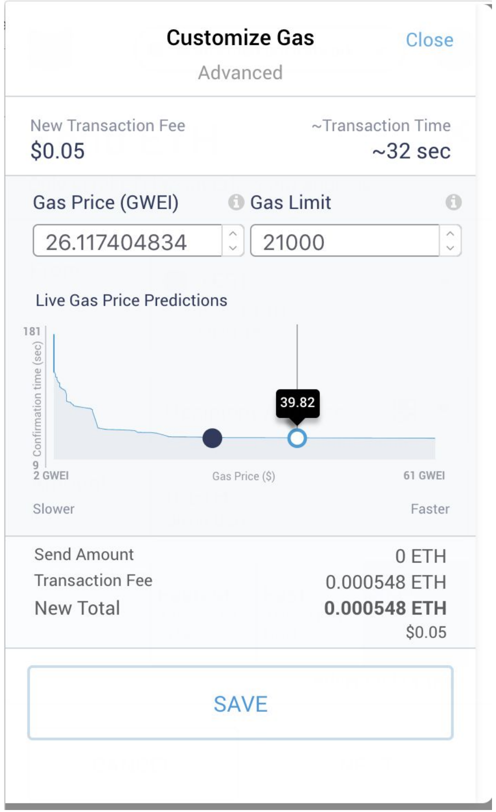
- The fee is proportional to the required amount of computation (EVM OPCODES)
- Each OPCODE has a costs named **gas**



Fees and gas

Each transaction costs a fee to the user

- Before each transaction, a user can set in their wallet:
    - The **gas price**: i.e. how much Ether they are willing to pay for each unit of gas
    - The **gas limit**: i.e. how many units of gas they are willing to consume for that transaction



Smart contracts: receive Ether

A function can be labelled as **payable** if it *expects* to receive Ether

- Once received the Ether the contract's balance is automatically increased, unless the transaction does not revert
- **msg.value** stores the received Ether (uint)

```
function foo() public payable {
    address payer = msg.sender; // Who sent the Ether
    uint received = msg.value; // How much *in wei*
    uint current = address(this).balance; // The current balance of the contract
}
```

Smart contracts: receive Ether

If a smart contract receives plain Ether, i.e. a transaction to the contract does not invoke a function:

- Trigger the **receive** function (>= Solidity 0.6.*)

If a transaction invokes a function that does not match any of the functions exposed by the contract, or as before but **receive** is not implemented:

- Trigger the **fallback** function

As before, but neither **receive** nor **fallback** are implemented

- Throws **exception**

Smart contracts: receive Ether

```
contract Example {
   // “address payable” labels an address meant to receive ETH from this contract
   address payable known_receiver;
   function forward() public payable {
       known_receiver.transfer(msg.value);
   }

   // All of them have in their body at most 2300 units of gas of computation available if
called by send() or transfer() (see next slide)
   receive() external payable {} // receive function
   fallback() external payable {} // fallback function Solidity >= 0.6.*
   function() public payable {} // fallback function Solidity < 0.6.*
}
```

Smart contracts: send Ether

If the contract has balance > 0, then it can send Ether as well

- Solutions that gives the receiver a gas limit of only 2300 units
  - **address.send(amount)** Send amount to *address*, returns True if everything goes well, False Otherwise
  - **address.transfer(amount)** Throws exception if it fails
- Solution with customizable gas limit
  - **address.call{options}(data bytes)** Returns True or False
  - *(bool result,) = address.call{gas: 123123, value: msg.value}("");*

Smart contracts: send Ether

Send/transfer: pros & cons

- A fixed gas limit prevents the receiver to execute too much code
  - It may consume too much gas to the original transaction sender
  - The receiver can execute malicious code, attempting an attack (e.g. reentrancy attack)
- Future updates to the gas associated to OPCODES (e.g. Istanbul fork) may break contracts already deployed working with limits of 2300 units of gas

https://consensys.net/diligence/blog/2019/09/stop-using-soliditys-transfer-now/

Smart contracts: events

It is possible to declare an **event** in Solidity similarly to a function, and it can be fired with the **emit** keyword

- Events are placed in the transaction log, useful for client apps

```
contract Example {
   event click();
   event executed(address sender);

   function press_click() public {
       emit click();
       emit executed(msg.sender);
   } }
```

References

Solidity documentation V 0.8.13: https://docs.soliditylang.org/en/v0.8.13/index.html

Accounts: https://ethereum.org/en/whitepaper/#ethereum-accounts

Sending Ether:

https://medium.com/daox/three-methods-to-transfer-funds-in-ethereum-by-means-of-solidity-5719944ed6e9

https://vomtom.at/solidity-0-6-4-and-call-value-curly-brackets/

Best practices: https://consensys.github.io/smart-contract-best-practices/

Data management: https://blog.openzeppelin.com/ethereum-in-depth-part-2-6339cf6bddb9/

Smart contracts: development

It is possible to implement Ethereum smart contracts with the Solidity programming language

Smart contracts can be developed and executed within:

- The browser IDE Remix, https://remix.ethereum.org/
- The CLI tool Truffle, https://www.trufflesuite.com/truffle

Extra

Advanced Solidity functionalities

Abi functions

The contract Abi (Application Binary Interface) is the standard contract-to-contract communication in Ethereum, to encode and decode functions, parameters, etc in known data, in bytes, to:

- Call a function of an external contract;
- Pass input arguments;
- And more.

https://docs.soliditylang.org/en/v0.8.4/abi-spec.html
https://docs.soliditylang.org/en/v0.8.4/units-and-global-variables.html#abi-encoding-and-decoding-functions

Abi functions

```
contract Decoder {

    function encodeArgs(uint _a, bool _b) public pure returns(bytes memory) {
        bytes memory data = abi.encode(_a, _b);
        return data;
    }

    function decodeArgs(bytes memory data) public pure returns(uint, bool)  {
        (uint _a, bool _b) = abi.decode(data, (uint, bool));
        return (_a, _b);
    }

}
```

Abi functions

```
contract HashContract {

    function encodeArgs(uint _a, bool _b) public pure returns(bytes memory) {
        bytes memory data = abi.encode(_a, _b);
        return data;
    }
    // The hash of arbitrary data can be computed with bytes32 hash =
kekkack256(abi.encode(param1, param2, ...));
    function computeHash(bytes memory data) public pure returns(bytes32) {
        bytes32 hash = keccak256(data);
        return hash;
    }
}
```

Calling contract functions

How to call a function of another smart contract?

- If you have the source code, you can **import** it on your Solidity file. Therefore, you have visibility of the contract's type and functions, and the compiler understands them
- If you DO NOT have the source code, you can use a low-level **call** to a function of a smart contract with the function's **selector** as input
  - The selector are the first 4 bytes of the hash of the function signature, i.e. *functionName(param1, param2, ...)*

Calling contract functions: import

```
contract External {

    uint public c;

    function increment() public {
        c = c + 1;
    }

    function increment(uint _a) public {
        c = c + _a;
    }

}
```

```
import "External.sol"
contract Caller {
    External contractExternal;
    constructor(address _c) public {
        contractExternal = External(_c);
    }
    function increment() public {
        contractExternal.increment();
    }
    function increment(uint _a) public {
        contractExternal.increment(_a);
    }
}
```

Calling contract functions: .call()

```
contract External {

    uint public c;

    function increment() public {
        c = c + 1;
    }

    function increment(uint _a) public {
        c = c + _a;
    }

}
```

```
contract Caller {
    address contractExternal;
    constructor(address _c) public {
        contractExternal = _c;
    }
    function increment() public {
        bytes4 selector =
bytes4(keccak256("increment()"));
        bytes memory data =
abi.encodeWithSelector(selector);
        (bool outcome, ) =
contractExternal.call(data);
        if(!outcome) revert(); } }
```

Calling contract functions: .call()

```
contract External {

    uint public c;

    function increment() public {
        c = c + 1;
    }

    function increment(uint _a) public {
        c = c + _a;
    }

}
```

```
contract Caller {
    address contractExternal;
    constructor(address _c) public {
        contractExternal = _c;
    }
    function increment(uint _a) public {
        bytes4 selector =
bytes4(keccak256("increment(uint)"));
        bytes memory data =
abi.encodeWithSelector(selector, _a);
        (bool outcome, ) =
contractExternal.call(data);
        if(!outcome) revert(); } }
```

Part 2
The Web3 library

An interface to interact with smart contracts

Web3

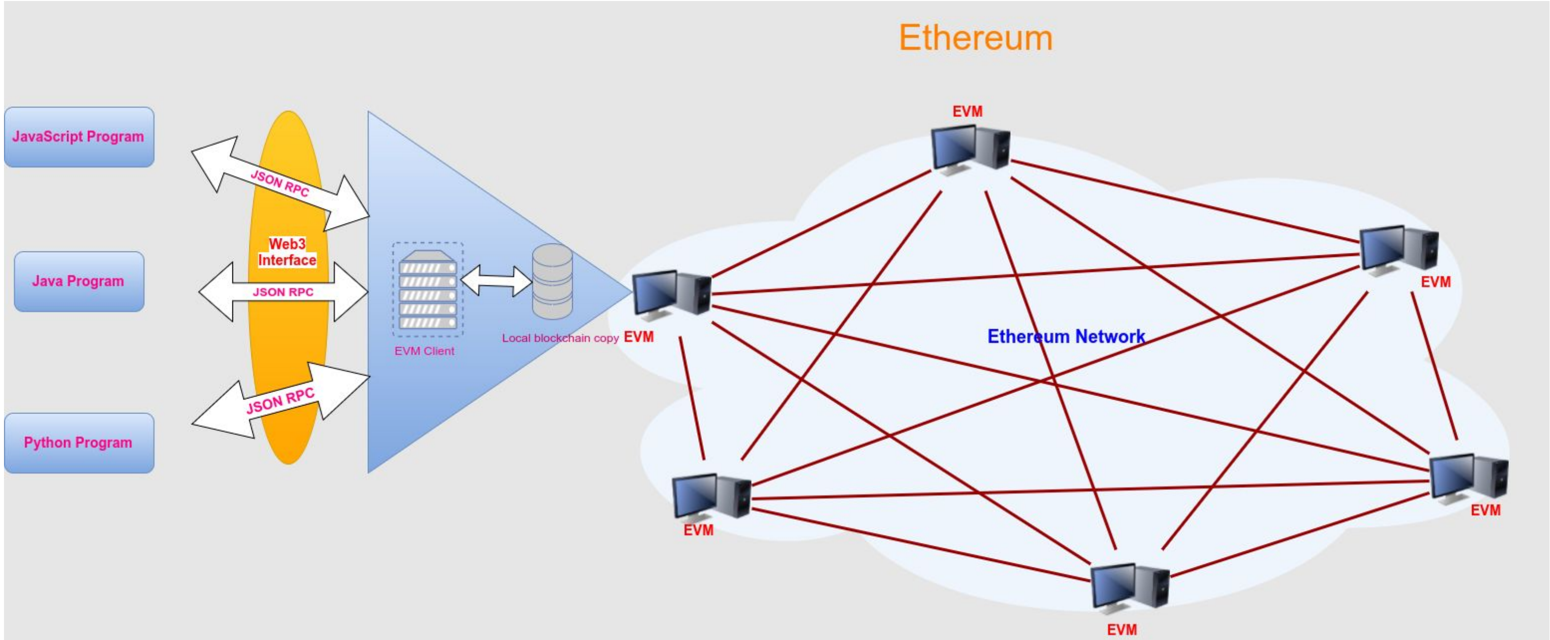
Web3 is a library to interact with the Ethereum network nodes with the RPC protocol, Remote Procedure Call

- Communications are asynchronous

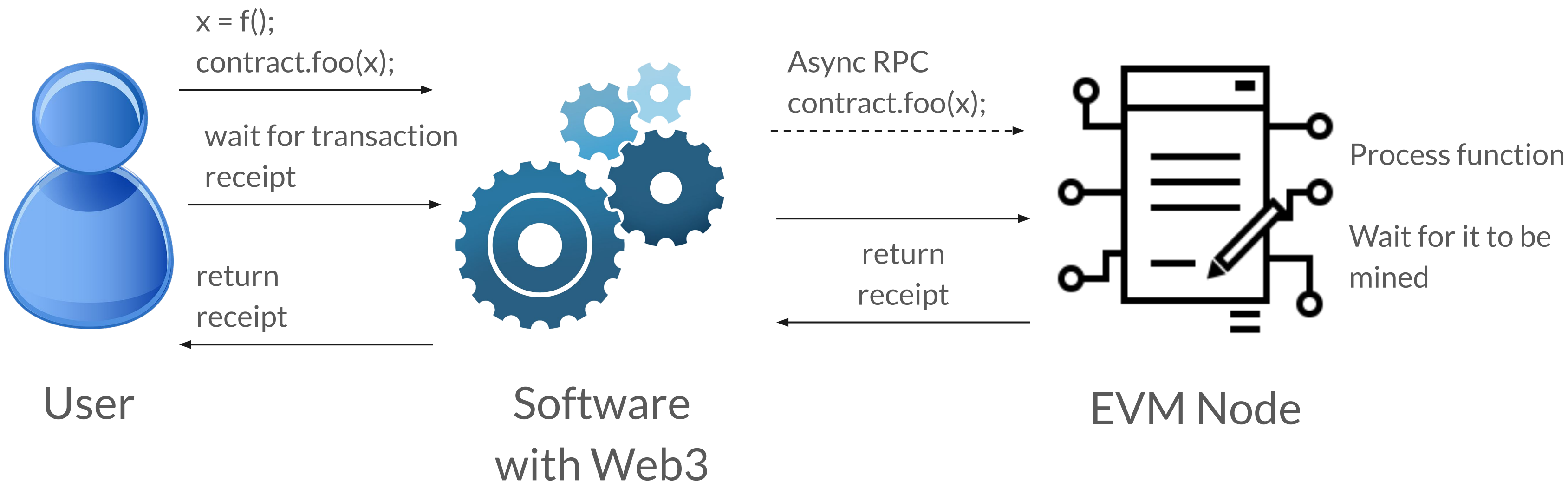
Software importing Web3 are able to communicate with smart contracts

Web3

Src: http://www.dappuniversity.com/articles/web3-js-intro



Web3


x = f();
contract.foo(x);
wait for transaction
receipt
return
receipt
Async RPC
contract.foo(x);
return
receipt
Process function
Wait for it to be
mined
User
Software
with Web3
EVM Node

Web3 implementations

[W1] **web3Js**: JavaScript

[W2] **web3J**: Java

[W3] **web3py**: Python

[W4] **web3.php**: Php

[W5] **hs-web3**: Haskell

NodeJs and Npm

In this tutorial we are going to use an environment based on Javascript
We need **NodeJs** and **Npm** (Node Package Manager)

Requirements: NodeJs

NodeJs is an environment to execute Javascript code on your machine instead on the browser:

- Write server-side Javascript code
- Modern frameworks for web development (ReactJs, AngularJs etc...)
- And Javascript desktop applications (ElectronJs)
- Install NodeJs
  - https://nodejs.org/en/docs/

Requirements: Npm

Npm (Node Package Manager) is the tool to install NodeJs packages

- Local packages are installed in the *./node_modules/* directory
  - Libraries and utilities for a single project
- Global packages are all installed in a single folder in your system
  - CLI tools to be reused among many projects
- It is installed with NodeJs
  - https://www.npmjs.com/get-npm
  - https://docs.npmjs.com/

References, Web3

[W1] Web3Js: https://github.com/ethereum/web3.js

[W2] Web3J: https://github.com/web3j/web3j

[W3] Web3Py: https://github.com/ethereum/web3.py

[W4] Web3.php: https://github.com/sc0Vu/web3.php

[W5] hs-Web3: https://github.com/airalab/hs-web3