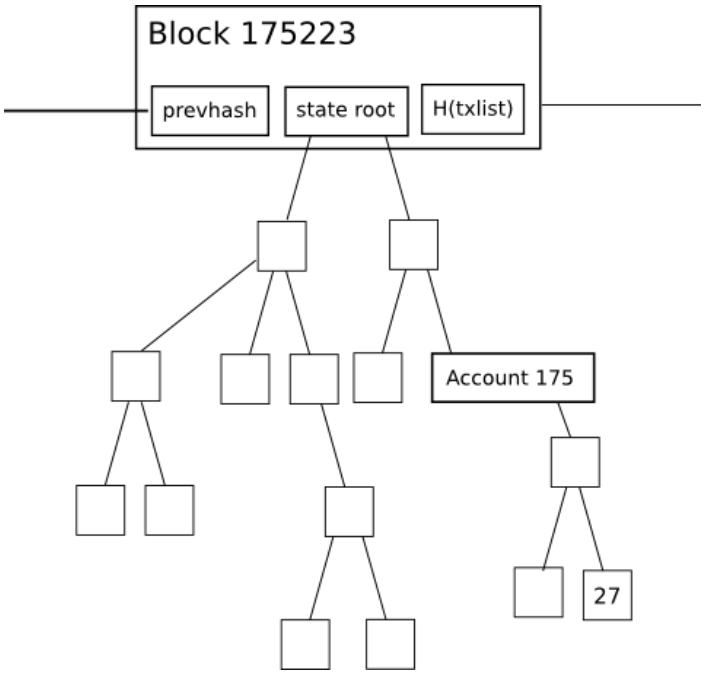
- ## Testnets

There are three various testnets currently in use and each behaves similarly to the production blockchain , where your real Ether and tokens reside. Developers may have a personal preference or favorite testnet, and projects typically develop on only one of them.

- **Ropsten**: A proof-of-work blockchain that most closely resembles Ethereum; you can easily mine faux-Ether.
- **Kovan**: A proof-of-authority blockchain, started by the Parity team. Ether can't be mined; it has to be requested.
- **Rinkeby**: A proof-of-authority blockchain, started by the Geth team. Ether can't be mined; it has to be requested.

One of the important issues is the large amount of data that clients are required to store; the amount of data in each Ethereum client's blockchain folder has ballooned to an impressive 10-40 gigabytes, depending on which client you are using and whether or not compression is enabled. Although it is important to note that this is indeed a stress test scenario where users are incentivized to dump transactions on the blockchain paying only the free test-ether as a transaction fee, and transaction throughput levels are thus several times higher than Bitcoin, it is nevertheless a legitimate concern for users, who in many cases do not have hundreds of gigabytes to spare on storing other people's transaction histories. There is no direct or fixed limit neither for transaction sizes nor for block sizes. This is a strength of the Etherem network, it does scale. That does not mean that there are no limits. There is the block gas limit. That means, in theory you could create a single transaction which consumes all the gas of a single block. First of all, the reason why the current Ethereum client database is so large is that Ethereum, unlike Bitcoin, has the property that every block contains something called the "state root": the root hash of a specialized kind of Merkle tree which stores the entire state of the system: all account balances, contract storage, contract code and account nonces are inside.

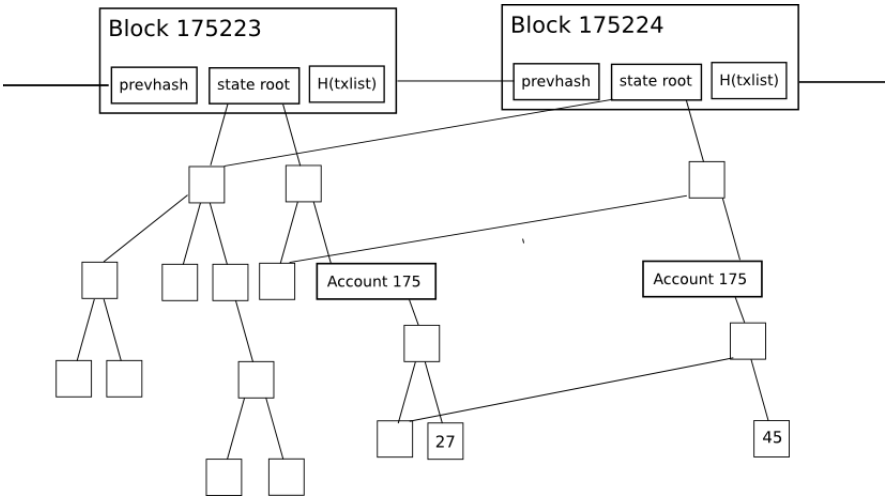


The purpose of this is: it allows a node given only the last block, together with some assurance that the last block actually is the most recent block, to "synchronize" with the blockchain extremely quickly without processing any historical transactions, by simply downloading the rest of the tree from nodes in the network (the proposed HashLookup wire protocol message will faciliate this), verifying that the tree is correct by checking that all of the hashes match up, and then proceeding from there. In a fully decentralized context, this will likely be done through an advanced version of Bitcoin's headers-first-verification strategy, which will look roughly as follows:

1. Download as many block headers as the client can get its hands on.
2. Determine the header which is on the end of the longest chain. Starting from that header, go back 100 blocks for safety, and call the block at that position $P^{100}(H)$ ("the hundredth-generation grandparent of the head")
3. Download the state tree from the state root of $P^{100}(H)$, using the HashLookup opcode (after the first one or two rounds, this can be parallelized among as many peers as desired). Verify that all parts of the tree match up.
4. Proceed normally from there.

For light clients, the state root is even more advantageous: they can immediately determine the exact balance and status of any account by simply asking the network for *a particular branch* of the tree, without needing to follow Bitcoin's multi-step 1-of-N "ask for all transaction outputs, then ask for all transactions spending those outputs, and take the remainder" light-client model.

However, this state tree mechanism has an important disadvantage if implemented naively: the intermediate nodes in the tree greatly increase the amount of disk space required to store all the data. In order to see why, consider this diagram here:



The change in the tree during each individual block is fairly small, and the magic of the tree as a data structure is that most of the data can simply be referenced twice without being copied. However, even still, for every change to the state that is made, a logarithmically large number of nodes (ie. ~5 at 1000 nodes, ~10 at 1000000 nodes, ~15 at 1000000000 nodes) need to be stored twice, one version for the old tree and one version for the new tree. Eventually, as a node processes every block, we can thus expect the total disk space utilization to be, in computer science terms, roughly O(n*log(n)), where n is the transaction load. In

practical terms, the Ethereum blockchain is only 1.3 gigabytes, but the size of the database including all these extra nodes is 10-40 gigabytes.

So, what can we do? One backward-looking fix is to simply go ahead and implement headers-first syncing, essentially resetting new users' hard disk consumption to zero, and allowing users to keep their hard disk consumption low by re-syncing every one or two months, but that is a somewhat ugly solution. The alternative approach is to implement *state tree pruning*: essentially, use reference counting so as to track when nodes in the tree (here using "node" in the computer-science term meaning "piece of data that is somewhere in a graph or tree structure", not "computer on the network") drop out of the tree, and at that point put them on "death row": unless the node somehow becomes used again within the next X blocks (eg. X = 5000), after that number of blocks pass the node should be permanently deleted from the database. Essentially, we store the tree nodes that are part of the current state, and we even store recent history, but we do not store history older than 5000 blocks.

X should be set as low as possible to conserve space, but setting X too low compromises robustness: once this technique is implemented, a node cannot revert back more than X blocks without essentially completely restarting synchronization. Now, let's see how this approach can be implemented fully, taking into account all of the corner cases:

1. When processing a block with number N, keep track of all nodes (in the state, tree and receipt trees) whose reference count drops to zero. Place the hashes of these nodes into a "death row" database in some kind of data structure so that the list can later be recalled by block number (specifically, block number N + X), and mark the node database entry itself as being deletion-worthy at block N + X.

2. If a node that is on death row gets re-instated (a practical example of this is account A acquiring some particular balance/nonce/code/storage combination f, then switching to a different value g, and then account B acquiring state f while the node for f is on death row), then increase its reference count back to one. If that node is deleted again at some future block M (with M > N), then put it back on the future block's death row to be deleted at block M + X.

3. When you get to processing block N + X, recall the list of hashes that you logged back during block N. Check the node associated with each hash; if the node is still marked for deletion *during that specific block* (ie. not reinstated, and importantly not reinstated and then re-marked for deletion *later*), delete it. Delete the list of hashes in the death row database as well.

4. Sometimes, the new head of a chain will not be on top of the previous head and you will need to revert a block. For these cases, you will need to keep in the database a journal of all changes to reference counts (that's "journal" as in journaling file systems; essentially an ordered list of the changes made); when reverting a block, delete the death row list generated when producing that block, and undo the changes made according to the journal (and delete the journal when you're done).

5. When processing a block, delete the journal at block N - X; you are not capable of reverting more than X blocks anyway, so the journal is superfluous (and, if kept, would in fact defeat the whole point of pruning).

Once this is done, the database should only be storing state nodes associated with the last X blocks, so you will still have all the information you need from those blocks but nothing more. On top of this, there are further optimizations. Particularly, after X blocks, transaction and receipt trees should be deleted entirely,

and even blocks may arguably be deleted as well - although there is an important argument for keeping some subset of "archive nodes" that store absolutely everything so as to help the rest of the network acquire the data that it needs.

Now, how much savings can this give us? As it turns out, quite a lot! Particularly, if we were to take the ultimate daredevil route and go X = 0 (ie. lose absolutely all ability to handle even single-block forks, storing no history whatsoever), then the size of the database would essentially be the size of the state: a value which, even now (this data was grabbed at block 670000) stands at roughly 40 megabytes - the majority of which is made up of with storage slots filled to deliberately spam the network. At X = 100000, we would get essentially the current size of 10-40 gigabytes, as most of the growth happened in the last hundred thousand blocks, and the extra space required for storing journals and death row lists would make up the rest of the difference. At every value in between, we can expect the disk space growth to be linear (ie. X = 10000 would take us about ninety percent of the way there to near-zero).

We may want to pursue a hybrid strategy: keeping every *block* but not every *state tree node*; in this case, we would need to add roughly 1.4 gigabytes to store the block data. It's important to note that the cause of the blockchain size is NOT fast block times; the block headers make up roughly 300 megabytes, and the rest is transactions of, so at high levels of usage we can expect to continue to see transactions dominate. That said, light clients will also need to prune block headers if they are to survive in low-memory circumstances. The root of the transaction tree that is unnecessary for each transaction to hold should be eliminated in order to reduce the size of Ethereum transactions effectively and optimizing the Ethereum blockchain. “Previously, each transaction’s receipt included the root of the transaction state tree. That is, the root of the Merkle tree immediately after that transaction was added to the tree. The state tree root, is one of the only dependencies one transaction has to other transactions in the block. By removing this dependency and adding several optional parameters, transactions can now be processed in parallel,” (Mohamed Abedelmalik). In the long-term, solutions like Plasma, which aim to deploy an interconnected network of blockchains wherein users are not required to verify every single point of data stored within the Ethereum blockchain network, will allow the Ethereum network to scale even more efficiently, handling larger daily transaction volumes and offering lower fees. Scalability is particularly important for Ethereum, arguably more so than bitcoin, because its transactions essentially represent fees users pay to decentralized applications launched on top of the Ethereum protocol. For instance, if a user is to use a LinkedIn-like business- and employment-oriented social networking application on Ethereum, the user requires to deploy a smart contract to the Ethereum blockchain in order to create a profile. To do so, since the deployment of smart contracts require transactions to be facilitated, gas or transaction fees are required to be sent by the user. If the transaction fee costs more than $1, it could become a problem for the user, especially if the application has millions of active users. Given Ethereum’s prioritization of flexibility and scalability, Ethereum co-founder Vitalik Buterin and the rest of the Ethereum Foundation have been preparing to migrate from proof-of-work (PoW) consensus protocol to proof-of-stake (PoS), or at least a hybrid system between the two, to increase the capacity of Ethereum. The capacity of the Ethereum blockchain has been increasing at a steady rate and the daily transaction volume of Ethereum has risen rapidly over the past few months.

Connecting to a testnet

Ethereum addresses & private keys that work on Ethereum, work on each testnet. Ether or tokens on the Ethereum main-net should not be sent to a testnet address in order not to lose your assets.Before proceeding, creating a new wallet specifically for use on testnets is strongly recommended; you’ll never accidentally send Ether, if you don’t have Ether. Utilizing MetaMask in order to send Ether and tokens on a testnet is straightforward; in the top-left of MetaMask, you can select an Ethereum network. Switch from the Main

Ethereum Network to Rinkeby (or other testnet) and you should see your balances and transaction history update so as to reflect the network you have selected. Now, when you create a transaction using MetaMask, it will be transmitted to the network you have selected.

1. The smart contract is executed by Ethereum virtual machine (EVM), so we always require a node or a piece of the Ethereum network in order to execute the contract.

```
$ node ./filename.js  -> okay for nodejs

$ go   ./filename.go  -> okay for golang

$ solc ./contract.sol -> not okay for solidity
```

Options to execute a contract

Here are some options to execute your contract:

1. Deploy the contract to **live (the real) Ethereum main network** and execute it.
2. Deploy the contract to **testnet (for developers' usage) Ethereum network** and execute it.
3. Deploy to an **Ethereum network simulator** and execute it.

In a development lifecycle which we usually want to do unlimited deploying, testing, updating,etc for many times. Then option 3 is the best option.

Installation packages

To prepare, do following commands to download truffle and ethereumjs-testrpc.

```
$ npm i -g truffle ethereumjs-testrpc
```

Prepare contract

Download this repo and go to 02-Testing/start

You should see the files like this

```
.
├── contracts
│   ├── HelloEthSalon.sol
│   └── Migrations.sol
├── migrations
│   └── 1_initial_migration.js
├── test
├── truffle-config.js
└── truffle.js
```

```
3 directories, 5 files
```

And the HelloEthSalon contract should look like this

```
pragma solidity ^0.4.4;

contract HelloEthSalon {
  string message = "Hello Ethereum Salon!";

  function HelloEthSalon() {
    // constructor
  }

  function GetMessage() returns (string) {
    return message;
  }
}
```

Testing file in Javascript

In the HelloEthSalon contract, we have a function GetMessage() that returns a string of Hello Ethereum Salon!

In the 02-Testingfolder, use truffle create test command to start a testing file.

```
$ truffle create test HelloEthSalon
```

As you can see, the new folder structure should be like this:

```
.
├── contracts
│   ├── HelloEthSalon.sol
│   └── Migrations.sol
├── migrations
│   └── 1_initial_migration.js
├── test
│   └── hello_eth_salon.js
├── truffle-config.js
```

```
└── truffle.js

5 directories, 8 files
```

We have a new file called hello_eth_salon.js under the test folder. The file looks like following,

```
contract('HelloEthSalon', function(accounts) {
  it("should assert true", function(done) {
    var hello_eth_salon = HelloEthSalon.deployed();
    assert.isTrue(true);
    done();
  });
});
```

Truffle smartly helps us naming the testing file as hello_eth_salon.js when we call the contract HelloEthSalon.

Testing file:

```
var HelloEthSalon = artifacts.require('./HelloEthSalon.sol');
contract('HelloEthSalon:GetMessage', function(accounts) {
  it("should return a correct string", function(done) {
    var hello_eth_salon = HelloEthSalon.deployed();
    hello_eth_salon.then(function(contract){
      return contract.GetMessage.call(); // **IMPORTANT
    }).then(function(result){
      assert.isTrue(result === 'Hello Ethereum Salon!');
      done();
    })
  });
});
```

Four important points:

1. Import the contract from './HelloEthSalon.sol' in the first line.
2. HelloEthSalon.deployed() is a Promise since interacting with Ethereum network is always an async behavior, which mean data will come back at some point later.
3. In line 7, use contract.GetMessage.call(); instead of using contract.GetMessage(); directly

4. The assertion statement assert.isTrue(result === 'Hello Ethereum Salon!'); is checking if the result we get from contract.GetMessage.call(); is equal to a string of Hello Ethereum Salon!.

Then, do the truffle test command in the 02-Testing folder,

```
$ truffle test ./test/hello_eth_salon.js
```

We might see the following common error:

```
Could not connect to your Ethereum client. Please check that your Ethereum client:
  - is running
  - is accepting RPC connections (i.e., "--rpc" option is used in geth)
  - is accessible over the network
  - is properly configured in your Truffle configuration file (truffle.js)
```

How to fix?

Start your Ethereum Simulator

When you do the truffle command, it tries to connect to an Ethereum node, and it, by default, tried to talk a local node on port localhost:8545 by a RPC manner, and notes 8545 is the conventional port for ethereum-node.

In distributed computing, a remote procedure call (RPC) is when a computer program causes a procedure (subroutine) to execute in a different address space (commonly on another computer on a shared network), which is coded as if it were a normal (local) procedure call, without the programmer explicitly coding the details for the remote interaction.

Eventually, Ethereum just follows the RPC approach to have a port that something can talk to. In order to start an ethereum-rpc on your machine, do command testrpc in a new terminal session

```
$ testrpc
```

and it will see it starts as normal:

```
EthereumJS TestRPC v6.0.3 (ganache-core: 2.0.2)


Available Accounts
==================
(0) 0x5cc6ee5c4fe0ff63dccb85d5faa802b4e42db244
(1) 0x2442e95edad609c8bf8952384eb1952ec46cc95c
(2) 0xf5f751a7b0fde6b2094bc61bbfe2ec3bc9ed4d80
(3) 0xdf1cf123bdd9871c0973d271045bc5ee138f055a
(4) 0x7a98259c8cb95b4f10e4425a44a2e5d3240b04a9
```

(5) 0xa1901a3f5f97615494b4605eb22a651de6dc00d8

(6) 0x2d0ada820168741f46ba15140da45d1f0e114296

(7) 0x5a8f3babe912170323b8f9bd725f67cfe2d8056a

(8) 0x3bde7be31bc39c4d8ad8d14ed7e23cbc4e92d3bd

(9) 0xaa3ab94e3205bdb32ebd2c0f5bccf240c1026fac

Private Keys

==========

(0) 565fc08465e8678ef34de22ba9d65d71f388992d04b9fc920e4f7bfa4e169639

(1) 86ebfebd22c626bb3d1ddf8c52332bea4b1f6c0ec210a23e7c248296609006ca

(2) 92b48fcae318e01c2fb082216df921c15b13a9d6842a624e269f2a5630ab63c4

(3) 1c053daa96d53e88585319b9ae1245bfcbc95c8fd8795ff470e98042e4537262

(4) de78e74db0a5c8dae8103c5990ccfed38ceb7b564b6ac6a2d478292af66cabea

(5) c66991438d785e225e9d785cd2e51c2e3adcafe50feb8fa77f6c670e32471f22

(6) 00f67f7649f94b73dacef7fd717dd125ff9f0be00762ca4e928d45885df65010

(7) 8407d5e870732f4cbf7d9ee78284db197fa95feffeec317bed294e1827231ffc

(8) 6800704a7ce0651271cb6392a3f477609f3c61abcec72cc0bf6a78e8c07f9b9c

(9) 09e13e69dad0025411039bc4e0e025b7d2b11aa5a46799318f2ae64fada6df42

HD Wallet

==========

Mnemonic: obtain cereal choose inquiry mercy perfect fee leopard owner live mechanic hip

Base HD Path: m/44'/60'/0'/0/{account_index}

Listening on localhost:8545

It will start with some fake accounts and private keys, and available on localhost:8545

In order to test it:

Do the truffle test command in the 02-Testing folder,

$ truffle test ./test/hello_eth_salon.js

First, you see it compile and then another Error occurs:

```
Using network 'development'.


Contract: HelloEthSalon:GetMessage
    1) should return a correct string
    > No events were emitted


0 passing (22ms)
  1 failing


1) Contract: HelloEthSalon:GetMessage should return a correct string:
     Error: HelloEthSalon has not been deployed to detected network (network/artifact mismatch)
      at /Users/andy/.nvm/versions/node/v8.5.0/lib/node_modules/truffle/build/cli.bundled.js:318327:17
      at <anonymous>
      at process._tickCallback (internal/process/next_tick.js:188:7)
```

The Error mentioned that HelloEthSalon has not been deployed to detected network (network/artifact mismatch).

Therefore, the thing is that besides starting the testrpc up, we also need to deploy our contract to the just-started testrpc Ethereum Simulator. What we need to do is to update our 1_initial_migration.js file under migrations folder. It should be looked

```
var Migrations = artifacts.require("./Migrations.sol");


module.exports = function(deployer) {
  deployer.deploy(Migrations);
};
```

Then import HelloEthSalon contract right after the MetaCoin importing statement and add a new deploy call right after the deployer.deploy(MetaCoin); . The updated deploy javascript file should be this:

```
var Migrations = artifacts.require("./Migrations.sol");
var HelloEthSalon = artifacts.require('./HelloEthSalon.sol');
```

```
module.exports = function(deployer) {
  deployer.deploy(Migrations);
  deployer.deploy(HelloEthSalon);
};
```

Again, do the truffle test command in the 02-Testing folder

```
$ truffle test ./test/hello_eth_salon.js
```

Firstly, you see it compile and do the testing as following:

```
Using network 'development'.


Contract: HelloEthSalon:GetMessage
    ✓ should return a correct string


1 passing (44ms)
```

If the ✓ checking mark appears, first smart contract testing is finished.

The whole testcase codes:

```
var HelloEthSalon = artifacts.require("./HelloEthSalon.sol");

contract("HelloEthSalon:GetMessage", function (accounts) {
  it("should return a correct string", async function () {
    const contract = await HelloEthSalon.deployed();
    const result = await contract.GetMessage.call();
    assert.isTrue(result === "I know smart contract testing!!");
  });
});
```

Change the contract and update the testing case.

Change the message global variable in the HelloEthSalon contract. Change the message to “I know testing of a contract!!”, And deploy it again by truffle deploy command.

```
$ truffle migrate
```

And go to update the test case, to check if it is the expected message.

```
$ truffle test ./test/hello_eth_salon.js
```

If the ✓ checking mark is seen, a safer contract by implementing testing is built.

The new contract should look like:

```
pragma solidity ^0.4.4;

contract HelloEthSalon {
  string message = "I know smart contract testing!!";

  function HelloEthSalon() {
    // constructor
  }

  function GetMessage() returns (string) {
    return message;
  }
}
```

And the updated testing file should look like

```
var HelloEthSalon = artifacts.require("./HelloEthSalon.sol");

contract("HelloEthSalon:GetMessage", function (accounts) {
  it("should return a correct string", async function () {
    const contract = await HelloEthSalon.deployed();
    const result = await contract.GetMessage.call();
    assert.isTrue(result === "I know smart contract testing!!");
  });
});
```

***

Use ‘async await’ to implement test; it makes the code that uses promises more readable.

Here is how the test look like with ‘async await’:

```
var HelloEthSalon = artifacts.require('./HelloEthSalon.sol');

contract('HelloEthSalon:GetMessage', function(accounts) {

it("should return a correct string", async function() {

var contract = await HelloEthSalon.deployed();

var result = await contract.GetMessage.call();

assert.isTrue(result === 'Hello Ethereum Salon!');

});

});
```

***For truffle develop (truffle 4), type migrate to deploy under the truffle develop session.

***Use artifacts to import HelloEthSalon

```
var HelloEthSalon = artifacts.require("./HelloEthSalon.sol");
```

***

- ## Test Smart Contracts on Ethereum

The Orbs blockchain relies on Ethereum in more than one way. At Orbs, we have an ERC20 token (ORBS) which among other things, can be utilized to purchase consumer apps oriented blockchain. Another application is subscription validation, which checks if a customer has a valid subscription to use the Orbs blockchain service.

In order to develop blockchain solution & check the subscription, a sidechain connector is required to Ethereum and connecting to Ethereum and more importantly testing the connection logic is not that simple.

Therefore Ethereum has a testnet of course (such as Ropsten). Although there is no actual cost, you still need to deploy a contract there and the process is not as simple as running an automated test—the testnet

still has some notion of resource limit. In more common terms, the testnet can be seen as a staging environment, but what about a development environment?

Various versions of blockchain have been built, such as blockchain utilizing TypeScript. However, there is another approach that would make it simple to test contracts and side-chain connector logic.

Truffle suit is a very broad feature set. Specific requirements are:

- Ensure the smart contract compiles
- Ensure we can call the smart contract function
- Make sure that the entire logic around the connection and object parsing is stable.

The following toolset is required to perform that:

- Ganache-core—part of the truffle suit, it is the Ethereum blockchain emulator
- Solc-js—based on the solc C++ project, it is a Javascript solidity compiler
- Web3.js—the Ethereum Javascript API

Remix tests solidity code, why go through all this trouble? And indeed remix is utilized to test the contract itself but eventually, although remix can run its own Javascript VM, we want to be able to test the code from Javascript implementation.

Exploring the web and looking on how people are using Typescript with the above libraries did not provide the solution of doing the above in a simple way. To be more specific, the truffle suit enables the features, but it feels over engineered. There are several wrappings of web3 for typescript, but that alone did not achieve goals and meet requirements.

Ethereum-simulator is based on Ganache core for blockchain simulation, solc to compile the contract and web3 to access the blockchain.

The main two use cases for this component can be:

- Testing a contract call from within Javascript/Typescript
- Testing your own web3-based rpc implementation vs. a Ganache instance

Using it to test a contract:

```
import {EthereumSimulator, EthereumFunctionInterface, EthereumFunctionParameter} from "ethereum-simulator"


async function example() {
    const sim = new EthereumSimulator();
    await sim.listen(8545);
    sim.addContract(contract);
    sim.setArguments(1, "whatever");
    const address = await sim.compileAndDeployContract();
```

```
    const storageFuncInterface: EthereumFunctionInterface = {
        name: "getValues",
        inputs: <EthereumFunctionParameter[]>[],
        outputs: [
            { name: "intValue", type: "uint256" },
            { name: "stringValue", type: "string" }
        ]
    };
    const data = await sim.callDataFromSimulator(address, storageFuncInterface);
    console.log(data.result["stringValue"]);
    sim.close();
}
```

The contract that fits this test is:

```
pragma solidity ^0.4.0;
contract SimpleStorage {
    struct Item {
        uint256 intValue;
        string stringValue;
    }
    Item item;

    constructor(uint256 _intValue, string _stringValue) public {
        set(_intValue, _stringValue);
    }

    function set(uint256 _intValue, string _stringValue) private {
        item.intValue = _intValue;
        item.stringValue = _stringValue;
    }
```

```
    function getInt() view public returns (uint256) {
        return item.intValue;
    }

    function getString() view public returns (string) {
        return item.stringValue;
    }

    function getValues() public view returns (uint256 intValue, string stringValue) {
        intValue = item.intValue;
        stringValue = item.stringValue;
    }
}
```

Breaking down the above, first define a new EthereamSimulator instance:

```
const sim = new EthereumSimulator();
await sim.listen(8545);
```

When calling listen(port) open the ganache core instance with an account that has more than enough 'ether' to do any operation required.

Then 'add' the contract and arguments to initialize it:

```
sim.addContract(contract);
sim.setArguments(1, "whatever");
```

These are synchronous operations which just load data into object, could have been a builder pattern if this was not so short, will perhaps change that if more functionality in the future is added.

The contract variable is the solidity contract as a string value.

After that compile and deploy the contract to the ganache instance, this will use solc to compile the contract and return the contract address after it was deployed.

```
const address = await sim.compileAndDeployContract();
```

It can take a couple of seconds for this line to execute, especially at the first time this contract is used (cold execution).

At this point the ganache has the contract ready and call and execute functions on this contract. The code below does that exactly using web3 behind the scenes:

```
const storageFuncInterface: EthereumFunctionInterface = {
  name: "getValues",
  inputs: <EthereumFunctionParameter[]>[],
  outputs: [
    { name: "intValue", type: "uint256" },
    { name: "stringValue", type: "string" }
  ]
};
const data = await sim.callDataFromSimulator(address, storageFuncInterface);
```

What returns from the callDataFormSimulator() call as ‘data’ in the example above is applying the DataFromEthereum interface:

```
export interface DataFromEthereum {
    result: { [key:string]: {value: string} };
    blockNumber: number;
    timestamp: number;
}
```

According to the function interface passed it will have an ‘intValue’ and ‘stringValue’ keys. At this example output the string value stored in contract and then close the simulator using:

```
console.log(data.result["stringValue"]);
sim.close();
```

In cases where you want to test with external code, you should defer calling the close() until your test code is done. For instance on how to use it with mocha and chai, you can refer to the package testing code that can be found in the github repo at test/ethereum-driver.spec.ts

This package makes it much simpler to setup and run the ganache-core code as well as test contracts from within system.

In the EthereumSimulator the function call callDataFromEthereum() will be displayed, which accepts an endpoint. This endpoint will be utilized in web3 to access the Ethereum node at that address. This was left open as a simple wrapper for accessing Ethereum and call a smart contract according to the interfaces that were built.

The library is available as an npm package at:https://www.npmjs.com/package/ethereum-simulator

Its code is available in Orbs’ GitHub repository at:

orbs-network/ethereum-simulator-typescript

ethereum-simulator-typescript - A typescript library to instrument ganache+web3+solc and test a smart contract…

github.com

What may be missing here:

The 0x project has an interesting project around web3 typescript wrapping and we can probably benefit working with the ABI and function calls should this package (ethereum-simulator) become larger and more complex.

No way to send() over Ethereum— As sending a transaction over Ethereum is an asynchronous operation, implementing such a feature for synchronous testing of a smart contract makes little sense, but still that does not mean this feature is redundant (and this is true for most of the other blockchains publicly available today).

The future releases of Orbs network will include a much wider support for cross-chain execution, especially from Ethereum. The ability to have both polyglot contracts and giving them the ability to access data on other blockchains is extremely important to enable applications to choose the right blockchain for the right data and use case. Testing frameworks enable us to move fast and achieve a quick feedback on how our code is behaving, and thus enable our blockchain customers better cross-chain abilities.

- ## Deploy Smart Contracts on Ethereum sketch:

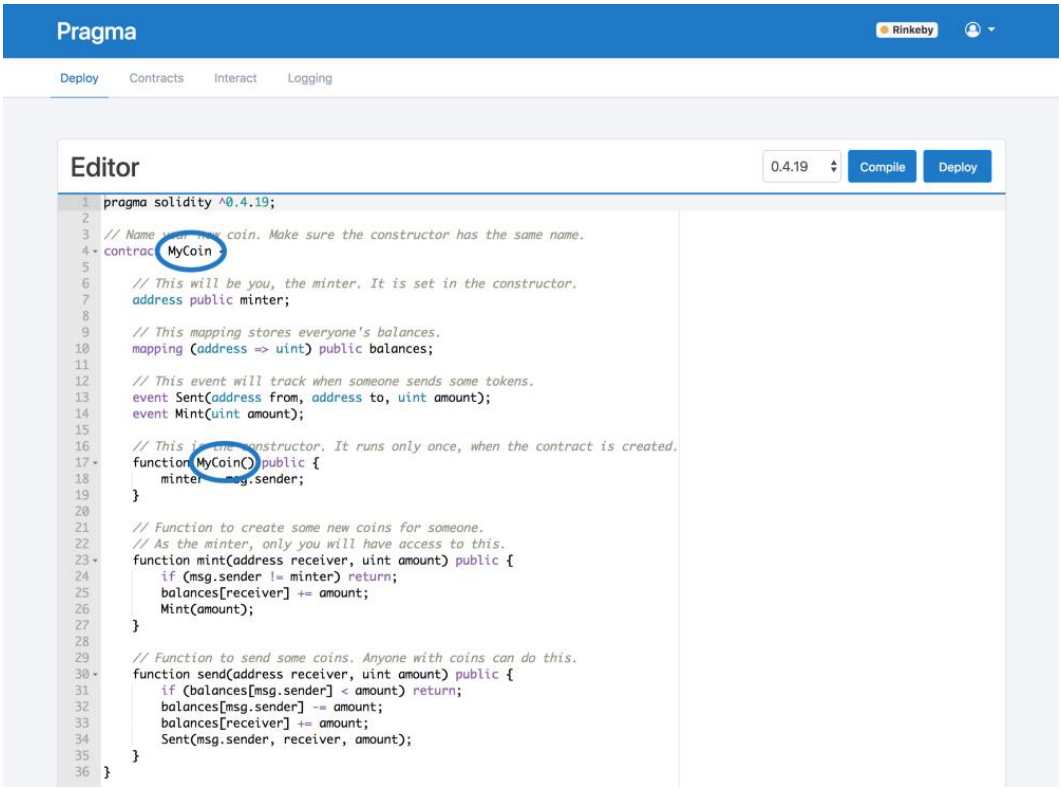
Install MetaMask

MetaMask is a browser extension that manages your Ethereum wallet and connects your browser to the blockchain. This connection enables you to send transactions to the Ethereum network. MetaMask is currently available for Chrome, Brave, Firefox, and Opera.

Follow the setup instructions to create a wallet. Make sure to safely store your mnemonic phrase and pick a secure password.

Craft a Contract

It’s easy to do this on Pragma, a web platform for Ethereum development. Head over to Pragma and make an account. Once you do, you can click into the Editor, shown below.



Pragma starts you off with a simple token contract. This isn't an official Ethereum token, but rather a more basic implementation. It's fine for this example though, so change the name of your coin to whatever you want, as seen on lines 4 and 17 in the picture above.

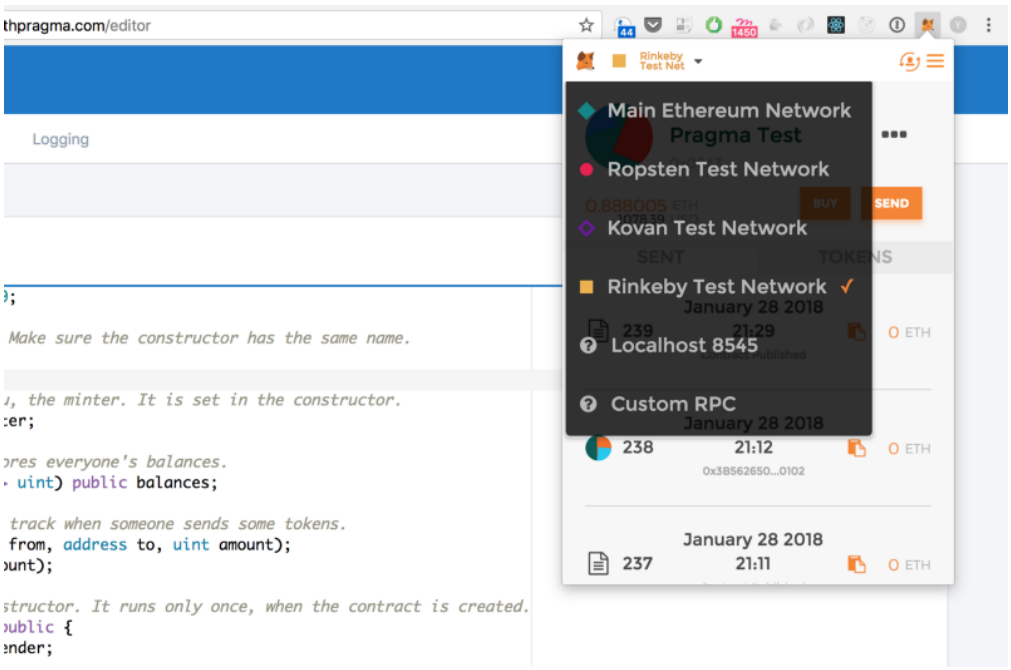
Get some Testnet Ether

In order to deploy contract and ETH .

There are also some test networks that you can use to test or debug your code.

In one of these test networks, called Rinkeby, where we can get some free test ETH.

Start by switching MetaMask over to the Rinkeby network, as shown in the picture below.



Then, use a website called a "faucet" that will send some Rinkeby test ETH into our account. Follow the instructions on the Rinkeby faucet website to get some.

This process isn't instant. It might take a while for the ETH to show up in your wallet, since there is a waiting period while the transaction is confirmed. Hopefully this will change in the future.

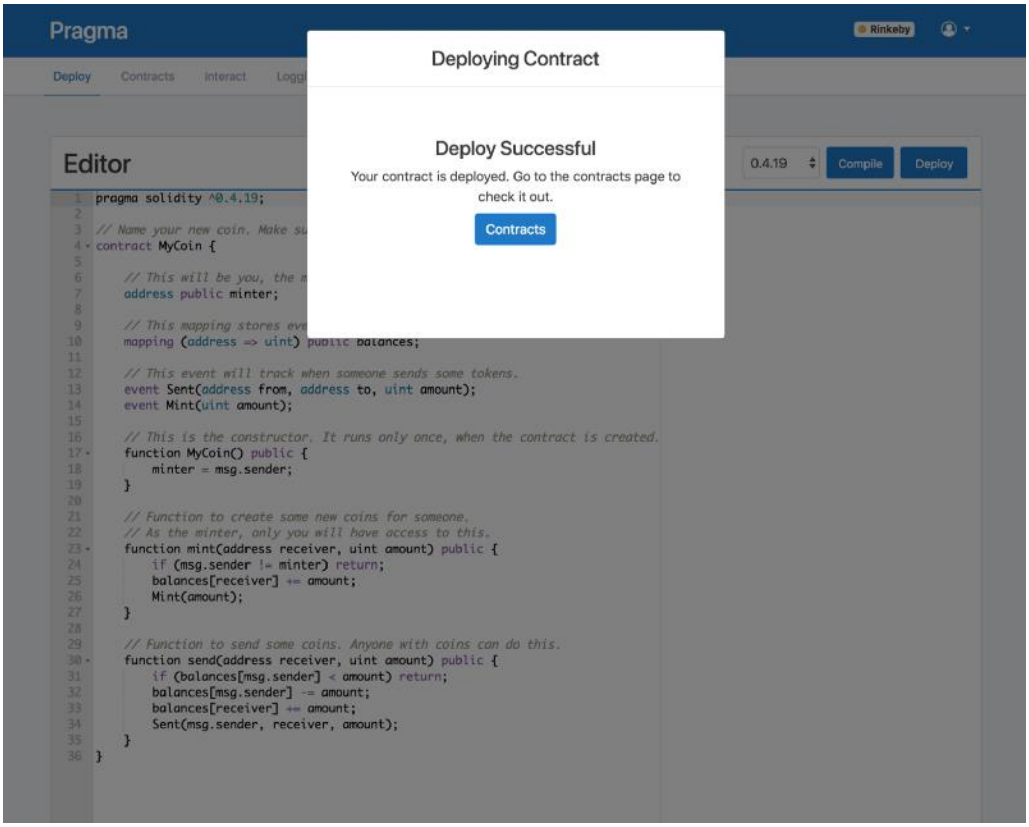
Once you've got the ETH in your MetaMask account, you're ready to go.

Deploy the Contract

The contract has these main parts:

1. Minter: This variable will be set to your Ethereum account address when the contract is deployed.
2. Balances: A mapping to keep track of how many coins each address holds.
3. Mint: A function that creates new coins. Only your account can call this function
4. Send: A function for sending coins from one account to another

To deploy your contract, click the deploy button in the top right of the editor. This will compile your contract and bring up a modal asking you to choose which contract to deploy. Since we only have one contract in the editor, your contract should already be selected in the input. Next, MetaMask will bring up a new window asking you to confirm your transaction. This is the step that actually deploys your contract code to the Rinkeby blockchain. Click the green submit button, and the progress modal in Pragma will move into the pending state. Your contract code has been submitted to the blockchain, and the transaction is waiting to be mined. Once your transaction is mined, you will see this screen.



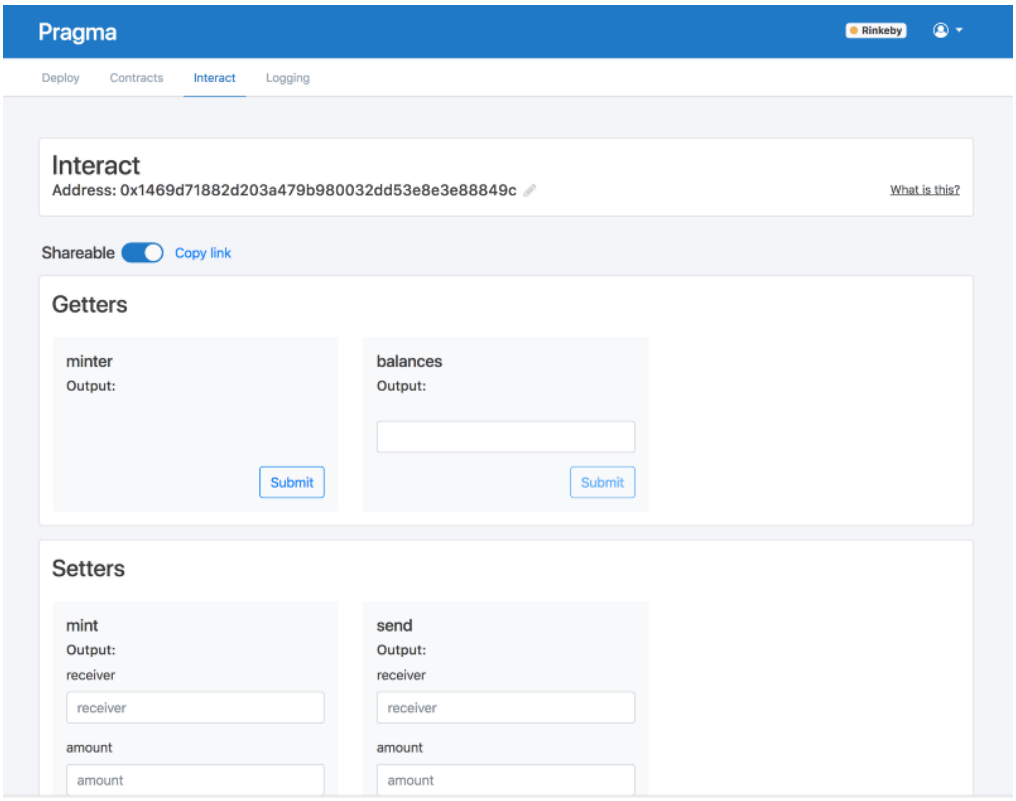
Now you're ready to start using your contract.

View your Contracts:

The contracts page lists all the contracts you've deployed before. You should see your contract in the table. From the actions dropdown on the right side, you can go to the logs for your contract, interact with your contract, edit your contracts details, or delete your contract from the list. It is important to note that deleting your contract from the list does not delete it from the blockchain. Once your contract is deployed to the Ethereum network, it is there permanently (unless you implement selfdestruct).

From the actions dropdown, click on the Interact option.

Interact with your Contract



A generated UI for your smart contract

The Interact page generates a UI that connects to your contract. The page is split into two sections: Getters and Setters. Getters allow you to read information from your contract. Setters allow you to write new information to your contract on the blockchain, which will trigger a transaction in MetaMask.

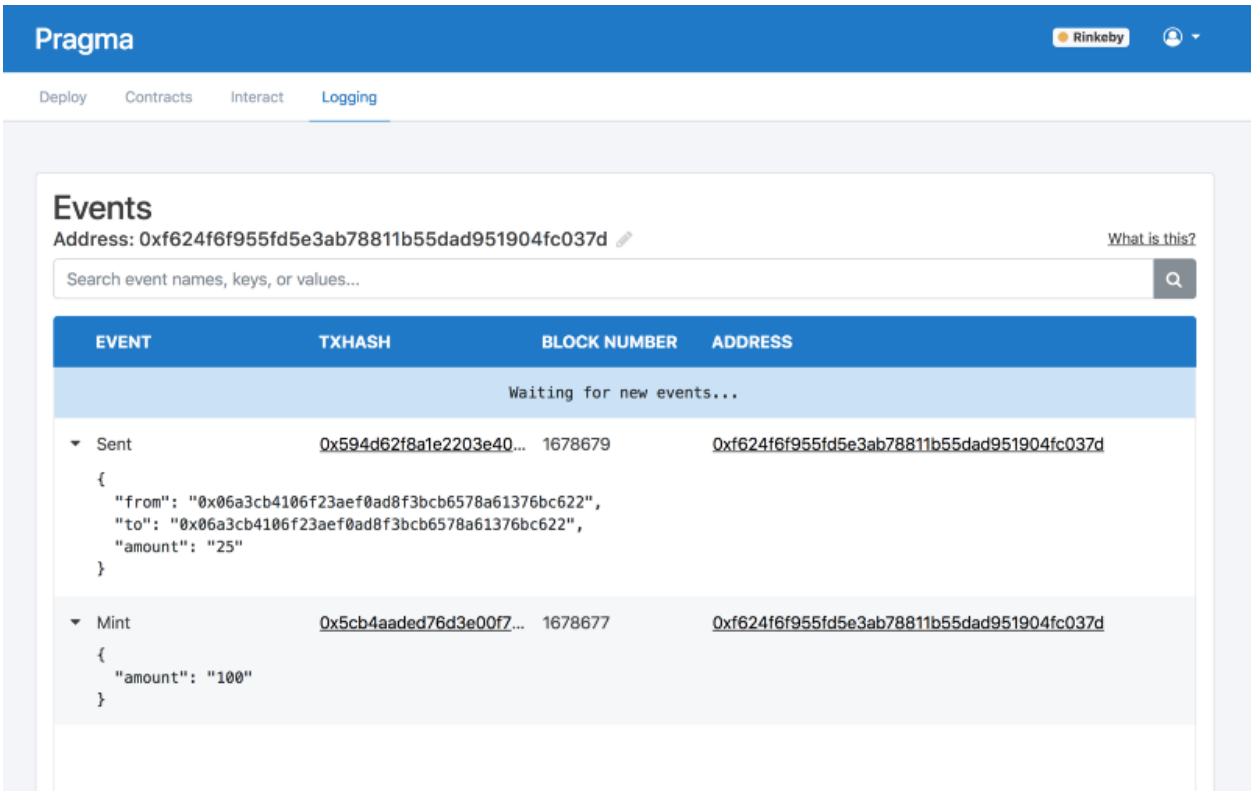
Here are some things to do with the contract

1. In the Getters section, find the "minter" function and click Submit. The output should be your own Ethereum address from MetaMask.
2. Copy your address and paste it into the input for "balances", and then click submit. You should see the value "0".
3. To give yourself some coins, you need to call "mint" in the Setters section. Paste your address into the "receiver" input, type 100 in the "amount" input, and click Submit. After confirming in a new MetaMask popup, your transaction is being mined. This step can take some time, but once it's finished, you can check your balance again and see the value 100.
4. To send some coins to other users, you can repeat the steps above in the "send" function.

Share your Contract

There's a toggle on the Interact page for making your contract shareable. If you turn this on, you'll get a link you can share wherever you want, which will allow other people to interact with your contract as well. This can be useful for testing your code, or just using a simple smart contract.

View Logs



Realtime logs view for smart contract

Every time the “mint” and “send” methods are called, an event from your contract will be logged. You can view all of the events from your contract in the logging view in Pragma. Click “Logging” in the subnav menu, and you should see one event for your minted coins. This logging view will load events as they happen in real-time.

- **Deploy Smart Contracts on Ropsten Testnet through Ethereum Remix sketch**

Deploy smart contract from the local blockchain—Ganache to Ropsten Test Net using the browser based IDE—Remix

Stage 1: Setup the Environment

Copy the election.sol (smart contract) and paste it into the remix IDE.

In order to deploy the contract, an account is required and with some ether on the Rospten test net.

Meta mask!

Select Rospten Test net

Create an Account

Copy Account Address

Request Free ether from https://faucet.metamask.io . Paste the copied account address in text box and click on send me 1 test ether.

We have the contract pasted in remix IDE, we have connected meta mask to the ropsten testnet with an account that has some ether.

Stage 2: Deploy the contract

Click on Run. Select Injected Web 3 Ropsten under environment and the account in Metamask is shown here under Account with balance ether as well.

Change the name of the contract to anything you like.

Click on create -> confirm transaction

Click on Metamask extension.

Click on the contract deployment—it should open up the etherscan page to look at our transaction details.

HODL

Stage 3: Interact with the contract

Pull / Access data from contract.

If you add 1 (candidate ID) near the candidates field, you should get the details of candidate ID 1. And 2 would give you the second candidate details.

Vote / add data to the chain.

After few seconds, the Metamask extension shows the transaction details. It should take you to ether scan if you click on it.

Transaction is successful at this stage.

If you copy address from metamask and paste it into etherscan, all the transactions within etherscan will be displayed. The INs will be the ethers that will come from the faucets. The OUTs will be spends. I have created the contract thrice and wrote on it once.

If you click on the first line item the vote transaction in etherscan the function vote will be called.

This is the easiest way to deploy your smart contract from a local blockchain environment (Ganache) to the Ropsten test net using the Ethereum Remix IDE.