Smart Contract Lab

1 Overview

A smart contract is a program that runs on a blockchain. Its code, which is immutable and stored at a particular address of the blockchain, gets executed automatically when a predetermined condition is met. Its data (state) is also stored on the blockchain. The objective of this lab is to help students understand how this type of program works and how to write simple smart contract code. Students will conduct experiments on the SEED blockchain emulator using the provided smart contract code. The following topics are covered in this lab:

- Smart contract development
- Remix IDE
- Deploy and interact with smart contract
- Send fund to and from smart contract

Lab environment. The activities in this document have been tested on our pre-built Ubuntu 20.04 VM, which can be downloaded from the SEED website.

2 Lab Setup: Starting the Blockchain Emulator

This lab will be performed inside the SEED Internet Emulator (simply called the emulator in this document). If this is the first time you use the emulator, it is important that you read this section. We recommend instructors to provide a lab session to help students get familiar with the emulator.

Download the emulator files. Please download the `Labsetup.zip` file from the web page, and unzip it. You will get the emulator files. The emulator consists of a number of container files, which are stored in the `Labsetup/emulator_*` folders. The `emulator_NN` folders are for AMD64 machines, while the `emulator_arm_NN` are for the Apple silicon machines. The number `NN` represents the number of nodes on the blockchain network: students can choose the smaller one if the RAM given for their virtual machine is less than 4GB.

To run the emulator, we only need these container files. These files are generated using the Python code stored in the `Labsetup/emulator_code` folder. Unless you want to modify the emulator files, you do not need to run the code in this folder (you need to install the SEED Emulator library from the GitHub to run the code). Instructors who would like to customize the emulator can modify the Python code and generate their own emulator files.

For the sake of simplicity, the blockchain running inside the emulator uses the Proof-of-Authority (PoA) consensus protocol, instead of the Proof-of-Stake protocol used in the MAINET. The activities conducted in this lab are not dependent on any specific consensus protocol.

Start the emulator. Go to the `emulator` folder, and run the following docker commands to build and start the containers. The commands listed below are aliases created on the SEED VM. If you are not using the SEED VM, you can use the original commands.

```
$ dcbuild        # Alias for: docker-compose build
$ dcup           # Alias for: docker-compose up
```

We recommend that you run the emulator inside the provided SEED Ubuntu 20.04 VM, but doing it in a generic Ubuntu 20.04 operating system should not have any problem, as long as the docker software is installed. For newer operating system version, the `docker-compose` command has already been phased out; it is integrated into the `docker` command, and you can run it using "`docker compose`", instead of `docker-compose`. Readers can find the docker manual from this link. If this is the first time you set up a SEED lab environment using containers, it is very important that you read the user manual.

All the containers will be running in the background. To run commands on a container, we often need to get a shell on that container. We first need to use the "`docker ps`" command to find out the ID of the container, and then use "`docker exec`" to start a shell on that container. We have created aliases for them in the `.bashrc` file.

```
$ dockps        // Alias for: docker ps --format "{{.ID}}  {{.Names}}"
$ docksh <id>   // Alias for: docker exec -it <id> /bin/bash

// The following example shows how to get a shell inside hostC
$ dockps
b1004832e275  hostA-10.9.0.5
0af4ea7a3e2e  hostB-10.9.0.6
9652715c8e0a  hostC-10.9.0.7

$ docksh 96
root@9652715c8e0a:/#

// Note: If a docker command requires a container ID, you do not need to
//       type the entire ID string. Typing the first few characters will
//       be sufficient, as long as they are unique among all the containers.
```

If you encounter problems when setting up the lab environment, please read the “Common Problems” section of the manual for potential solutions.

Stop the emulator. To stop the emulator, we just need to stop all the containers. We can go to the terminal where we run the "`docker-compose up`" command, type `Ctrl-C`. That will stop all the containers, but without removing them, i.e., all the data in the containers are still preserved, and they can be resumed by running "`docker-compose up`" again. If we want to remove them, we can run the "`docker-compose down`" command. Another way to do this is to go to a different terminal (but still in the same `emulator` folder) and directly run this command. That will stop and removing all the containers.

```
$ dcdown        # Alias for: docker-compose down
```

EtherView. We have implemented a simple web application called `EtherView` to display the activities on the Blockchain. To access the application, point your browser (within the VM) to `http://localhost:5000/`. From the `Blocks` page, you can see the newly created blocks and recent transactions. If nobody is sending transactions, the blocks are mostly empty, i.e., containing no transactions.

Once we start sending transactions, we should be able to see them. Users can click on the blocks and transactions to see their details.

3 Task 1: Using Remix for Smart Contract Development

There are several development environments for smart contracts, including Hardhat, Remix, and Truffle. In this lab, we will use Remix IDE (Integrated Development Environment), which can be used to write, compile, and debug the Solidity code. It supports testing, debugging and deploying smart contracts.

3.1 Task 1.a: Connecting Remix to the SEED Emulator

Remix IDE has an online version and a desktop version. We will use the online IDE, so there is no need to install anything. Simply go to this URL `https://remix.ethereum.org/`, and you will get the Remix IDE. Remix provides excellent instructions on how to use the IDE at this URL `https://remix-ide.readthedocs.io/`.

Remix can connect to blockchain via several mechanisms, such as Remix VM, injected provider (MetaMask), and external HTTP providers. The Remix VM is sandbox blockchain in the browser; it simulates the behavior of a blockchain, so it is not a real blockchain. We will connect Remix to our SEED blockchain emulator, which is a real blockchain (a private blockchain). This can be done using an injected provider (such as MetaMask) or an external HTTP provider.

We will use MetaMask in this task. We first connect MetaMask to the SEED blockchain emulator, and then tell Remix to use MetaMask whenever it needs to interact with the blockchain. MetaMask not only serves as our bridge to the blockchain, it is also a wallet: whenever we need to send a transaction, we have to do that from an account; we use MetaMask to manage our accounts.

In the Blockchain exploration lab, we have already installed MetaMask extension to our browser. If you have not done that lab before, you can follow the instruction in this URL to set it up:`https://github.com/seed-labs/seed-labs/blob/master/manuals/emulator/metamask.md`. To restore the accounts, please use the following mnemonic phrase:

```
gentle always fun glass foster produce north tail security list example gain
```

After setting up MetaMask, go to the Remix IDE, in the `"Deploy & Run Transactions` menu, set the `Environment` to `"Injected Provider - MetaMask"`. If everything is set up correctly, in the `Account` drop-down menu, you should be able to see several accounts with balance.

3.2 1.b: Write, Compile, and Deploy Smart Contract

In this task, we will use Remix to deploy a simple smart contract called `Hello.sol`. The code can be found in the `Labsetup/contract` folder.

- *Writing code*: Go to the `Workspaces` menu in Remix, and you will see several folders. Inside the `contract` folder, create a new file called `Hello.sol`. Copy and paste the content of the provided `Hello.sol` file to this new file.

- *Compiling code*: Go to the `"Solidity compiler"` menu, and compile the `Hello.sol` program.

- *Deploying contract*: Go to the `"Deploy & run transactions"` menu and click the `Deploy` button. MetaMask will come up and ask you to confirm. If you do not see MetaMask, check your

`Environment` setting to make sure that MetaMask is selected. If the deployment is successful, you should be able to see the confirmation from MetaMask. Go to EtherView and show that your deployment transaction has been added to a block.

3.3 1.c: Under the hood

In this task, we will see what really happens when we deploy a contract. Deploying a contract is done through a transaction, which is different from fund-transfer transaction. Please deploy a contract, and then use EtherView to get the transaction details. Compare this transaction with a fund-transfer transaction, and describe the main differences between these two types of transactions. In the contract-deployment transaction, you will see some content in the data field contain. What is this content? Please provide evidence to support your conclusion.

Note: In this task, you may need to get the bytecode of a compiled contract. This can be obtained from Remix. In the `"Solidity Compiler"` menu, you should be able to see the `ABI` and `Bytecode` buttons. You can also go to the `Labsetup/contract` folder, compile the code using the provided `Makefile`. You can get the bytecode.

4 Task 2: Invoke Contract Functions

In this task, we will see how to interact with a smart contract. We will use the `Hello` contract deployed from the previous task. Go to the `"Deployed Contracts"` region, and we should be able to see the contracts that were just deployed, including their addresses, balances, and the functions.

4.1 Task 2.a: Invoke a function via local call

If a function is defined as a `"public view"` type, it does not modify the state of the contract, and is thus invoked through a local call, instead of via a transaction. There is no cost for this type of interaction. Remix creates a button for this type of function, as well as for the `public` variable (for each `public` variable, a default getter function of the `"public view"` type is created). We can interact with the contract using these buttons. Please invoke all the `"public view"` functions of the contract, and report your observation.

4.2 Task 2.b: Invoke a function via transaction

If a function modifies the sate of the contract, it has to be invoked via a transaction. Remix uses a different color for the buttons corresponding to these functions. Please invoke the `increaseCounter()` function, and report your observation. Please uses EtherView to show the transactions generated by this invocation.

Add a new function called `decreaseCounter(unit)` function to the smart contract. Compile and Deploy it. Interact with this new function and show your results.

4.3 Task 2.c: Under the hood

What has actually happened when we send a transaction to invoke a function? Please take a look at the `to` and the `data` fields of such a transaction, and explain their meanings. To invoke a function, we need to provide the function name and arguments. This information is encoded in the the `data` field. Please invoke the `increaseCounter()` function, get the transaction details using EtherView, and then verify whether the `data` field is indeed encoded based on the following:

- Function selector: The first 4 bytes of the hash of the function's prototype
- Arguments: encoded according to the types defined in ABI (32 bytes each)

The following Python script help you calculate the hash of a function's prototype.

```
from web3 import Web3

hash = Web3.sha3(text="<function signature>")
print(hash.hex())
```

After understanding what really happens, let us invoke a function directly through the data field, instead of using the buttons provided by Remix (those buttons actually construct the content for the data field). In Remix, at the bottom of the `"Deployed contract` region, you can see a region called `"Low level interactions"`. Whatever we put in the `CALLDATA` field will be put in the `data` field of the transaction. Please use this method to invoke the `increaseCounter()` function, and verify that the invocation is successful.

4.4 Task 2.d: Emit events

Smart contracts can emit events to communicate that something has happened on the blockchain. Applications can listen to these events and take actions when they happen. In `Hello.sol`, we have declared an event called `MyMessage`. Then in the `sendMessage()` function, we use `emit` to generate a message.

```
event MyMessage(address indexed _from, uint256 _value);

function sendMessage() public returns (uint256) {
   emit MyMessage(msg.sender, counter);
   return counter;
}
```

The generated messages are placed inside the log field of a transaction receipt. By monitoring this field, an application can be notified. Remix does display the log field. Please take a look at the field, and confirm that its content is what you have expected.

5 Task 3: Send Fund to Contract

In this task, we will send fund to a contract. We will use a different contract for this task. The contract program is called `EtherFaucet.sol`, and it can be found in the `Labsetup/contract` folder. Please copy and paste the Solidity source code to Remix, compile it, and then deploy it to the SEED blockchain.

When we send fund to a smart contract, some of the contract's function (must be the `payable` type) will be invoked. Depending on the situation, different functions will be invoked. Figure 1 depicts the conditions for invoking each function. In this task, we will conduct experiments for each of the situations.

5.1 Task 3.a: Send fund directly to a contract address

One can directly directly send fund to a contract using the contract's address. This is done through a regular transaction. In this case, we are not specifying any function to invoke in our transaction. However, when a contract receives the fund, one of its function will be invoked.

Please use MetaMask to send some fund to the `EtherFaucet` contract using its address. Once the transaction has been confirmed, check the balance of the smart contract on Remix. The `amount` variable should contain the same value as the balance (in different unit). Conduct the following experiments:

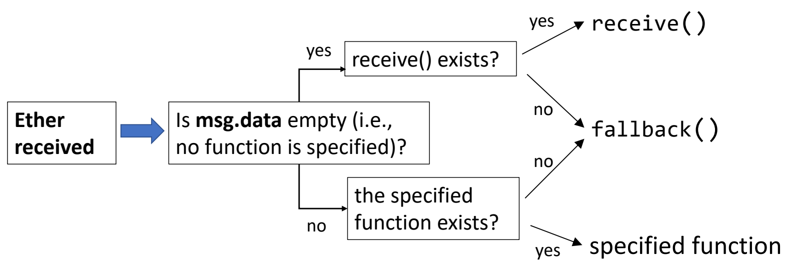

Figure 1: Which payable function is invoked when fund is sent to a contract

- Check the `count` variable of the contract to verify that the function invoked is consistent with what is described in Figure 1.
- Try to remove the `receive()` function from the contract, and see what warnings Remix will be giving.

5.2 Task 3.b: Send fund to a payable function

We can send fund to a contract while invoking a payable function. We will do this from Remix and invoke the `donateEther()` function to send some Ethers to the contract. How much is sent to the contract is specified in the `value` field of the transaction. You need to put some amount in this field (which is displayed above the `Deploy` button). The function invoked can get the fund amount using `msg.value`. After the invocation, please verify that the values of the balance, the `amount` variable and the `donationCount` variable are what is expected.

```
function donateEther() external payable {
    amount += msg.value;
    donationCount += 1;
}
```

5.3 Task 3.c: Send fund to a non-payable function

Function that can receive funds must be `payable`. Please try to remove the `payable` keyword from the `donateEther()` function and see what warnings Remix will be giving. We also provided a function called `donateEtherWrong()`, which is not `payable`. Please send some fund to the contract via invoking this function. You may receive some warnings, but do ignore them and send out the transaction. What will happen? Will the fund transfer succeed? Will the sender lose money if it does not succeed?

```
function donateEtherWrong() external {
    donationCount += 1;
}
```

5.4 Task 3.d: Send fund to a non-existing function

In this task, we will send fund to a contract while invoking a function. However, we intentionally make a mistake by invoking a function that does not exist in the contract. We will see what will actually happen.

To invoke a non-existing function, we will directly set the `data` field of our transaction. As what we did in 4.3, we can use the "`Low level interactions`" of Remix to directly set the `data` field. Let us invoke a function called `foo()`, which does not exist in the contract. We will send some fund to the contract while invoking this function. Please show what happens. Will the fund sending successful? What function will be invoked? etc.

6 Task 4: Send Fund from Contract

In the previous task, we see how to send fund to a contract. In this task, we will see how to send fund from a contract to a recipient. In Solidity, there are three different methods for sending ether from a contract, including `send`, `transfer`, and `call`. All three methods are translated into the CALL opcode by the Solidity compiler, so they share the same internal process, but they have differences. Regarding which one is recommended to use, the existing documentation on the Internet has been quite confusing because of the constant changes of the Ethereum blockchain.

The major difference among these three methods is the gas limit forwarded to the recipient. When a transaction is sent, a gas limit is set by the original sender. When this transaction invokes a contract A, which triggers the contract to send ether to a recipient, if the recipient is also a contract (say B), some of B's code will be executed. B needs some gas to run, and how much gas it gets depends on how much is forwarded by the sender contract A.

The `send` and `transfer` methods set a limit of fixed 2300 gas forwarded to the recipient, With this limited gas, B cannot do much. The main reason for placing such a limit is to prevent the reentrancy attack (which is covered by another SEED lab). The `call` by default does not have such a limit, i.e., it forwards all the gas set by the original sender. We can also set a limit when using `call` method.

Due to the changes of the blockchain, the gas costs of the code might change, and the 2300 gas limit might not be sufficient in the future. This may cause some smart contract applications to break. Therefore, suggestions have been made to not to use the `send` and `transfer` method. Instead, use the `call` method. Although the `call` method is subject to the reentrancy attack, there are many ways to prevent such an attack. Actually, after the Istanbul hardfork, the `send` and `transfer` methods have been deprecated.

Task: Send fund to an external owned account. We have implemented the send-fund functionality in our `EtherFaucet` contract using all these three methods. Please use Remix to invoke each of these methods, and see whether you can receive ether from the contract. It should be noted that the `_amount` argument uses the wei as its unit, so if we want to get one ether, we should use `1e18` as the amount.

```
function getEtherViaCall(uint256 _amount) external payable {
    address payable to = payable(msg.sender);
    amount -= _amount;
    (bool success, ) = to.call{value:  _amount}("");
    require(success, "Failure: Ether not sent");
}

function getEtherViaSend(uint256 _amount) external payable {
    address payable to = payable(msg.sender);
    amount -= _amount;
    bool success = to.send(_amount);
    require(success, "Failure: Ether not sent");
}

function getEtherViaTransfer(uint256 _amount) external payable {
    address payable to = payable(msg.sender);
    amount -= _amount;
    to.transfer(_amount);
}
```

It should be noted that in this sample code, everybody can get any amount of ether from this contract, so this is definitely not secure. In a real smart contract, access control should be placed to ensure only authorized accounts can receive fund from this contract. Also, for the sake of simplicity, we did not put any protection against the reentrancy attack in the code. We have a separate SEED lab on the reentrancy attack.

7 Task 5: Invoke Another Contract

Inside a contract, we can invoke the functions of another contract. In this task, we will see how such an invocation works. We have provided a smart contract called `Caller.sol`, which can be found from the `Labsetup/contract` folder. We will use it to invoke the functions in the `Hello.sol` contract, which was deployed in Task 1.

The code snippet is listed below. It needs to include an `interface` derived from the `Hello` contract. This way, we can create a contract object, and then use the APIs in the interface to interact with the `Hello` contract. We have provided two invocation examples. The only difference is that one only does the invocation, and the other also sends fund to the `Hello` contract during the invocation.

```
function invokeHello(address addr, uint _val) public returns (uint) {
   Hello c = Hello(addr);
   uint256 v =  c.increaseCounter(_val);

   emit ReturnValue(msg.sender, v);
   return v;
}

function invokeHello2(address addr, uint _val) public onlyOwner
                                               returns (uint)
{
   Hello c = Hello(addr);
   uint256 v =  c.increaseCounter2{value: 1 ether}(_val);
                                             ↖ send fund
   emit ReturnValue(msg.sender, v);
   return v;
}

interface Hello {
    function sayHello() external pure returns (string memory);
    function getResult(uint a, uint b) external view returns (uint);
    function increaseCounter(uint k) external returns (uint);
    function increaseCounter2(uint k) external payable returns (uint);
    function getCounter() external view returns (uint);
    function sendMessage() external returns (uint256);
}
```

Task. Please deploy the `Caller` contract, and invoke its `invokeHello()` and `invokeHello2()` functions. Please describe your observation and explain what you see.

8 Submission

You need to submit a detailed lab report, with screenshots, to describe what you have done and what you have observed. You also need to provide explanation to the observations that are interesting or surprising. Please also list the important code snippets followed by explanation. Simply attaching code without any explanation will not receive credits.