Bogazici University

Integration of IoT and Blockchain

Project Report

Room Monitoring with Arduino using NodeRED on top of Ethereum

Author:
Mehmet Doğan

Supervisor:
Arda Yurdakul

September 28, 2018

1 Motivation

The world will have 50 billion connected devices by 2020 according to predictions of trusted resources like IBM, Cisco [1]. As 50 billion IoT devices come online, the industry will face some challenges, such as ensuring the security of its devices, guaranteeing the privacy of the device owners, preventing loss of valuable data, improving the scalability of cloud services, and handling all the resulting data garbage. In other words, these challenges based on reliability, security, privacy, and scalability. All of these main challenges are pointed out in blockchain technology. Some blockchain infrastructures are anonymous, so it resolves privacy issue. They are completely decentralized, so it resolves reliability by eliminating the risk of Single Point of Failure. By using off-chain solutions, it can scale M2M transactions up to millions of transactions with solutions like Lightning Network, sharding, mass-validate and can serve the spread of autonomous systems via smart contracts.

In this report, we present the implementations and results of a use-case study which is a Proof-Of-Concept for Internet of Things and blockchain technologies. We designed an embedded system which is detailed in section 2.1. We developed the processing software by using Node-RED, which is an IoT Design Framework. By using Node-RED, we prevented to reinvent the well and reduced development costs. Using Node-RED enables end device to communicate with Ethereum Network and Swarm.

Ethereum is a decentralized software platform that enables smart contracts and Distributed Applications (DApps) to be built and run without any downtime, fraud, control or interference from a third party. Ethereum is not just a platform but also a programming language (Turing complete) running on a blockchain, helping developers to build distributed applications [2].

Smart contracts are a series of instructions and behave just like Finite State Machines. Smart contracts are executed in Ethereum Virtual Machine which is mostly written in Solidity language, yet other alternative languages can be compiled to bytecode and executed on EVM. Solidity is mostly preferred because of security concerns because it works on IF-THIS-THEN-THAT logic and has lots of restrictions. Basically, it is condition based language and only after the first set of instructions are done, the next function can be executed. Then, it continues just like in a sequence until it reaches the end of the contract [3].

Complementary technology to blockchain is distributed file storage systems and decentralized databases like Swarm, IPFS and BigchainDB, respectively. These technologies mostly based on distributed hash tables (DHT). In this experiment, we decided to go on with Swarm which is one of the major projects under Ethereum Foundation. Swarm is a distributed storage platform and content distribution service, a natural base layer service of the Ethereum web3 stack. The primary objective of Swarm is to provide a sufficiently decentralized and redundant store of Ethereum's public record, in particular, to store and distribute Dapp code and data as well as blockchain data [4]. It's important to note that none of these systems have proven the stability including Swarm.

In addition to hardware environment, we used Geth client(version 1.6.7) of Ethereum. To write and deploy smart contracts, we used Remix IDE which can be run on the browser. The contract written in Solidity generates two components: the bytecode to run on EVM and the Application Binary Interface (ABI). Bytecode runs whenever a function is called from the application, and stored into Ethereum blockchain under contract address. ABI defines the structures and functions that can be invoked explicitly. In other words, ABI grants access to call functions in smart contracts. To sum up, three requirements should be satisfied to interact with a smart contract: 1) Bytecode must be deployed to blockchain 2) Address of bytecode must be known 3) ABI of smart contract must be known.

In summary, the outcomes of the experiment are as follows:

- Design concept of an end-to-end embedded system
- Basic principles of blockchain and decentralized storage technologies
- Creating transactions on Ethereum network
- Writing smart contract and designing decentralized autonomous application (dApp)
- Communicating with smart contracts by using web3 library of Node.js
- Pushing IoT device data into the Swarm file system and storing reference to the file in smart contract

2 Setting up the Environment

2.1 Hardware Environment

An embedded system sense the environment by taking data through sensors, process data coming from sensors to generate the desired output to the system from actors. Connectivity makes the embedded system to communicate with the other connected devices. We use the light-dependent resistor (LDR) and microphone as the sensors. LDR sensor is used in sensing light. The resistance of LDR varies according to the amount of light that falls on it. Microphone sound sensor module is used as a sound sensor in this hands-on session. Our sensor module can be used in sound alarm systems. Processing of sound and light data is done on a do-it-theself (DIY) board, i.e., Arduino. We develop the processing software by using Node-RED, which is an IoT Design Framework. We put a BLE module on Arduino to get our embedded system connected with our smartphone.

LDR Sensor gives an output voltage which is directly proportional to the light intensity on it. It is connected to an analog input pin on the Arduino. The reading from LDR is the voltage input referenced to 5V and expressed as a 10-bit value (Therefore, range of the value is 0-1024). One leg of LDR is connected to 5V and the other one to analog input pin A0 on Arduino. You need to create a voltage divider mechanism as shown in Figure 1.

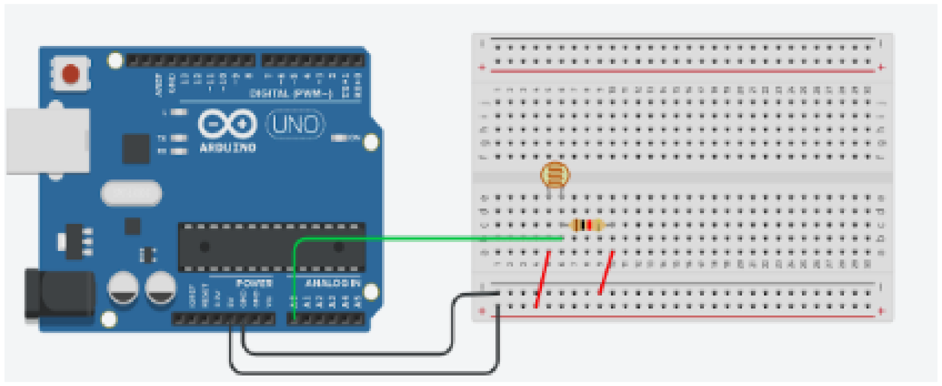


Figure 1: Connection of LDR and Arduino to Breadboard

To interact with Arduino, there are several ways. We will use Firmata Protocol which provides direct access to IO pins and also compatible with Node-RED. Standard Firmata Library is found in Arduino IDE under

Files → Examples → Firmata → Standard Firmata

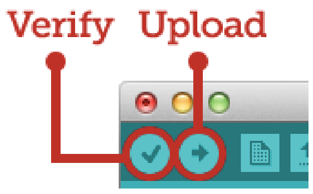


Figure 2: Arduino IDE verify the code and upload to Arduino

On Node-RED palette as shown in Figure 3, we have following items in order : Arduino IN - Arduino OUT. The node with symbol on the left is Arduino IN.

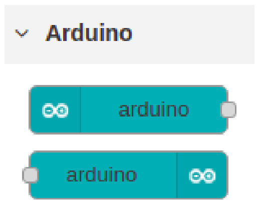


Figure 3: NodeRED Arduino input and output nodes

We upload Standard Firmata Library to Arduino by clicking verify and upload on the top left, respectively. After connection and installation is completed, we determine light density threshold under normal conditions. In order to do this, we create the Node-RED flow that consists of three nodes:

- **LDR:** Since LDR is connected to one of the inputs of Arduino, we put an "Arduino-IN" node by dragging-and-dropping from the palette to the canvas.
  - Set Arduino port properly
  - Set input type as analog
  - Set PIN number as 0
  - Set Name as *LDR*

- **Interval:** Set “Delay” node’s parameters as “Rate Limit” for “all messages” as “1 message per 5 second” and ‘Drop Intermediate Messages’.
- **LDR Value**: To see the output add a ”Debug” node.

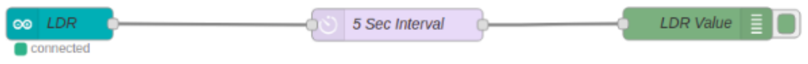


Figure 4: Node-RED flow to obtain data from LDR sensor

We add an microphone sound sensor module to the breadboard. Sound sensor detects the sound intensity of the environment. It has both analog and digital output:

- AO: analog output, real-time output voltage signal of the microphone
- DO: digital output. When the sound intensity passed a certain threshold, the output is high

We use digital output (DO), because we need to capture whether there is noise or not.

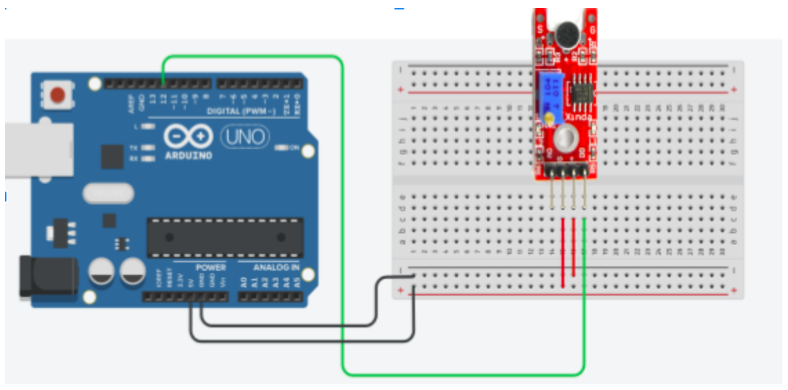


Figure 5: Connection of Microphone Sound Sensor and Arduino

After wiring the component to breadboard, we need to use “Switch” node in this part. Because on detection of noise like a clap, the payload of the message should be 1, meaning there is a noise during a fixed period time. Therefore, we add a “Delay” node to the output of this node where there is silence. Otherwise, sensor reading will turn back to 0, immediately. We set “Delay” node as “Delay each message” with “Fixed Delay” as 15 seconds.

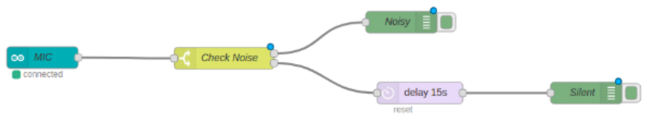


Figure 6: Connection of Microphone Sound Sensor and Arduino

L298N motor driver will be used as the dimmer for controlling light. We will generate PWM (Pulse Width Modulation) signal as the dimming signal from Arduino. We start with connecting Arduino and L298N by following the diagram as shown in Figure 7 and table as shown in Figure 8.

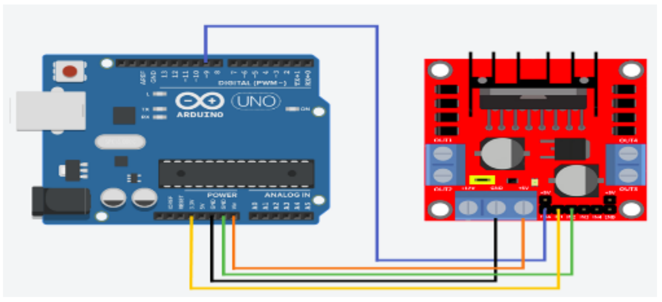
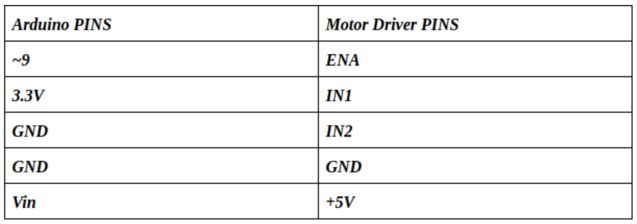


Figure 7: Connection Diagram of L298N and Arduino



Arduino PINS	***Motor Driver PINS***
~9	***ENA***
3.3V	***IN1***
GND	***IN2***
GND	***GND***
Vin	***+5V***

Figure 8: Correponding PINs for Arduino and L298N connection

For testing the effect of dimming, we will change the duty cycle of PWM. As the duty cycle increases, lamp will shine more. We can achieve this by using 3 nodes:

- **Timestamp:** Drag-and-drop 'inject' node. It will display as "timestamp". If it does not display like that, then double-click the node and drop-down "payload" menu to find the timestamp. You need to set "repeat" interval as every 2 seconds.

- **Change:** Add "function" node to the flow. Name it as "change". Timestamp will always pass its current value to "change". We will apply modulo on timestamp to bound the accumulated timestamp value. Note that this modulo is the Tperiod of the PWM. Accumulated timestamps will change Ton value. You can access the timestamp by calling "msg.payload". Payload is the timestamp of current time in milliseconds since January 1, 1970. So, your Function block should look like as follows:

```
        msg.payload = Math.round(msg.payload/100 % 255)
        return msg;
```

- **Lamp:** From palette select "Arduino OUT" node and add it to flow. The output of "change" node will be the input to the lamp. It expects an analogue input. Therefore, we select type as "analogue, and the pin as 9 (Recall that we connected Motor Driver to pin 9). Lastly, set the name of "Arduino OUT" node.

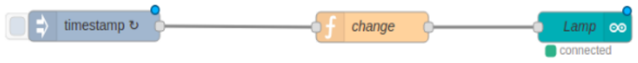


Figure 9: NodeRED flow for dimming light

In order to make actors and sensors work together, we need to make sure that sensor inputs are combined and controlled before actors working. In Node-RED, aside from the "msg" object, the "function" can also store data between invocations within its "context" object. We use this specific "context" object in Node-RED to get the sensor inputs and concatenate them. In the function "Concatenate", the sensor values from LDR and microphone are added to the context according to their "msg.topic" (which defines the IDs of the sensors). Then these values will be packed into one message payload. In the function "turnLight", we will write a simple if/else statement which controls the light density and microphone sensor value. If the room is dark, the light will turn on. If the room's light density below a threshold and there is a noise, the light will turn on as well, but with less luminosity. Otherwise, the light will be off. The flow of the system is shown in Figure 10.

We will start with implementation of "Concatenate" function, which combines readings from two different sensors. To achieve this, we need to use

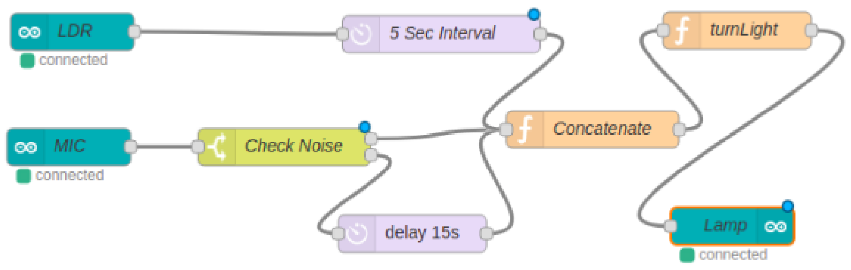


Figure 10: NodeRED flow of acting according to sensors

"Context" object in node-RED. To initialize context, do the following which sets the value to 0 if it is null:

```
    context.ldr = context.ldr || 0.0;
  context.mic = context.mic || 0;
```

Then, we need to set this values according to sensor readings. The "msg" object in node-RED also includes "msg.topic" in addition to msg.payload. You can see msg.topic by connecting a "Debug" node to LDR sensor. By default, msg.topic is the pin number for arduino-In nodes.

```
    if (msg.topic === 'A0') {
        context.ldr = msg.payload;
    } else if (msg.topic === 'A12') {
        context.mic = msg.payload;
    }
    msg = {
        payload: {
            "LightSensorVal" : context.ldr,
            "MICSensorVal" : context.mic
        }
    };
    return msg;
```

Final part in this section is to implement "TurnLight" function. Your device must autonomously respond to environmental conditions. Therefore, you will create a condition-oriented function. You will use if/else if/else structure on this purpose. You can access the LDR value by msg.payload.LightSensorVal and MIC value by msg.payload.MICSensorVal (Recall that we set both in JSON object above).

```
    if(msg.payload.LightSensorVal< 100  ){
        msg.payload= 255; // Most intense light
    }
    else if (msg.payload.LightSensorVal< 300  && msg.payload.
       MICSensorVal===1) {
        msg.payload= 150;  // Dimmed light
    }
    else{
        msg.payload= 0;    // No Light
    }
    return msg;
```

2.2 Ethereum and Swarm Setup

Geth is the the command line interface for running a full Ethereum node implemented in Go [5].By installing and running Geth, you can take part in the Ethereum frontier live network and

- Ethereum mining by solving hash puzzles
- Transfer funds between addresses
- Deploy contracts and send transactions
- Explore block history

2.3 Installing Go and Dependencies

For building Geth from source, Go and C compilers are need to be installed. Therefore, we create an empty folder in the home directory.

Create new directory and change directory with following commands.

```
mkdir ~/homeMonitor
cd ~/homeMonitor
```

Download necessary packages to install Go. Then, download Go version 1.10.1 via curl. Unpack it and export environment variable.

```
sudo apt-get update
sudo apt install -y curl
sudo apt install -y build-essential
sudo apt install -y screen
```

```
curl -O https://dl.google.com/go/go1.9.1.linux-amd64.tar.gz
sudo tar -C /usr/local -xzf go1.9.1.linux-amd64.tar.gz
export PATH=$PATH:/usr/local/go/bin
```

We check the installation is succesfully completed.

```
go version
```

If it's installed properly, the output will be like **go version go1.10.1 linux/amd64** and can remove archive file from the current directory. we need to repeat above steps carefully.

```
rm go1.9.linux-amd64.tar.gz
```

2.4 Building Swarm and Geth

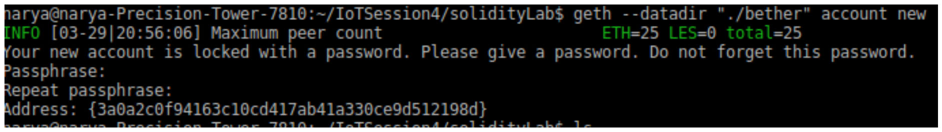
Then, we install ethereum source code and build swarm and geth with the following commands.

```
sudo apt-get install git
git clone https://github.com/ethereum/go-ethereum.git
cd go-ethereum
git checkout v1.6.7
make geth
sudo cp build/bin/geth /usr/local/bin/geth
make swarm
sudo cp build/bin/swarm /usr/local/bin/swarm
```

After ensuring that the installation is completed and all dependencies are built properly, we

- Create a new wallet on our private Ethereum Network,
- Execute the init command which initializes the definition for the network and start synchronization,
- Assure that connection is successful to e-node (Peercount must be 1 when **net** command is run) by carrying out the following steps:

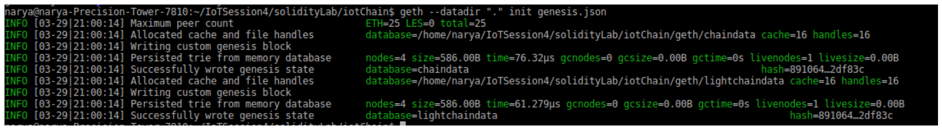
```
cd ~
geth --datadir "./iotChain" account new
```



```
narya@narya-Precision-Tower-7810:~/IoTSession4/solidityLab$ geth --datadir "./bether" account new
INFO [03-29|20:56:06] Maximum peer count                       ETH=25 LES=0 total=25
Your new account is locked with a password. Please give a password. Do not forget this password.
Passphrase:
Repeat passphrase:
Address: {3a0a2c0f94163c10cd417ab41a330ce9d512198d}
```

Figure 11: Expected output of of $ geth account new

```
cd iotChain
geth --datadir "." init genesis.json
geth --rpc --rpcport "8000" --rpccorsdomain "*" --datadir "."
    --port "30303" --nodiscover --rpcapi "db,eth,net,web3,
    personal" --identity "one" --networkid 666 console
net
```



```
narya@narya-Precision-Tower-7810:~/IoTSession4/solidityLab/iotChain$ geth --datadir "." init genesis.json
INFO [03-29|21:00:14] Maximum peer count                       ETH=25 LES=0 total=25
INFO [03-29|21:00:14] Allocated cache and file handles         database=/home/narya/IoTSession4/solidityLab/iotChain/geth/chaindata cache=16 handles=16
INFO [03-29|21:00:14] Writing custom genesis block
INFO [03-29|21:00:14] Persisted trie from memory database      nodes=4 size=586.00B time=76.32µs gcnodes=0 gcsize=0.00B gctime=0s livenodes=1 livesize=0.00B
INFO [03-29|21:00:14] Successfully wrote genesis state         database=chaindata                                                  hash=891064…2df83c
INFO [03-29|21:00:14] Allocated cache and file handles         database=/home/narya/IoTSession4/solidityLab/iotChain/geth/lightchaindata cache=16 handles=16
INFO [03-29|21:00:14] Writing custom genesis block
INFO [03-29|21:00:14] Persisted trie from memory database      nodes=4 size=586.00B time=61.279µs gcnodes=0 gcsize=0.00B gctime=0s livenodes=1 livesize=0.00B
INFO [03-29|21:00:14] Successfully wrote genesis state         database=lightchaindata                                                  hash=891064…2df83c
```

Figure 12: Expected output of $ geth init genesis.json

After complete synchronization, we also need to connect swarm file sharing system. To achieve this, from geth console, we get the address by executing **eth.coinbase**. It returns the address of the Ethereum wallet. Now, we need to open another terminal to run the swarm node. Then, we change to directory which includes geth and keystore. We crop "0x" part from the address and export it as the BZZKEY with following commands.

```
  export BZZKEY=theCOINBASEADDRESS
  swarm --bzzaccount $BZZKEY --datadir "." --ens-api "./geth.ipc
      " --bzznetworkid 666 --bootnodes "enode://21091
      bdbb66b87660edacb29e45a210b4391e5911ca4a2d13c4440c7136
fae87be3e1a05ddd0b4c2dd1c737c0a558621f49c337b0102c9af1
b4ab6bbce822632@67.207.75.210:30399"
  curl -s -L http://localhost:8500/bzz:/73bc757953aac456639
   2f453579f6126f83cc1dc960690f2359c05bfb7cb02d1
```

2.5 Web3 Installation and Node-RED Configuration

Web3.js is a collection of libraries which allow you to interact with a local or remote ethereum node, using a HTTP or IPC connection [6]. To interact with Ethereum via Node-RED, we be using web3. Assuming that, we have node package manager(npm) which should be installed with 'node.js'. Change directory to node-RED, where it is installed in home directory for Ubuntu users.

```
cd ~/.node-red
npm install --save body-parser
npm install --save ethereum.js
npm install --save express
npm install --save web3@0.19
```

Following that, open **settings.js** with the favorite editor and modify functionGlobalContext as below

```
functionGlobalContext: {
      web3:require('web3')
  },
```

3 Getter and Setter on Solidity

In this part, we implement a basic smart contract that stores an integer and getter&setter methods for this value. You need to open the favorite browser and go to http://remix.ethereum.org. Create new contract:

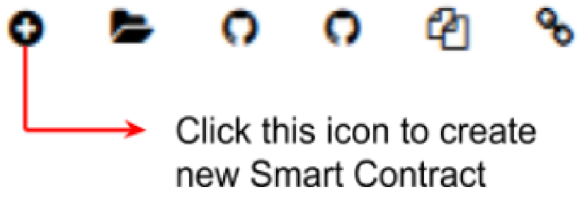


Figure 13: Remix IDE Online: Top Left Menu

we start with declaring compiler version of solidity which we write code. Then, with **contract** keyword, we define the contract.

```
pragma solidity ^0.4.20;
contract simpleContract{
  \\the code goes here
}
```

Listing 1: smart contract Outline

You need to review the code below, and try to understand the syntax, key-words. we be asked to write the own contract for the later tasks.

```
pragma solidity ^0.4.20;
contract simpleContract{
    uint x; // an unsigned integer
    // The keyword "public" makes this variable readable from
        outside.
    function setData(uint _x) public { // Sets the value of x
        x = _x;
    }
    // The keyword "view" cannot change the state of EVM.
    // The keyword "returns" means that the function will return
         values.
    function getData() public view returns(uint){ //
        return x;
    }
}
```

Listing 2: A simple smart contract for storing a value, ‘x‘

Now, you have a simple smart contract that is ready to deploy to network. When you compile the code on IDE, it will make statistical code analysis. Therefore, it will generate warnings or errors. To block these warnings and errors, you can deactivate ‘Auto Compile’ option.

By clicking details, you can get meta-data, ABI, byte-codes, assembly and more about the contract. **As mentioned earlier, you are required to have ABI of the contract for accessing it from application**. ABI basically describes all the functions in the contract.

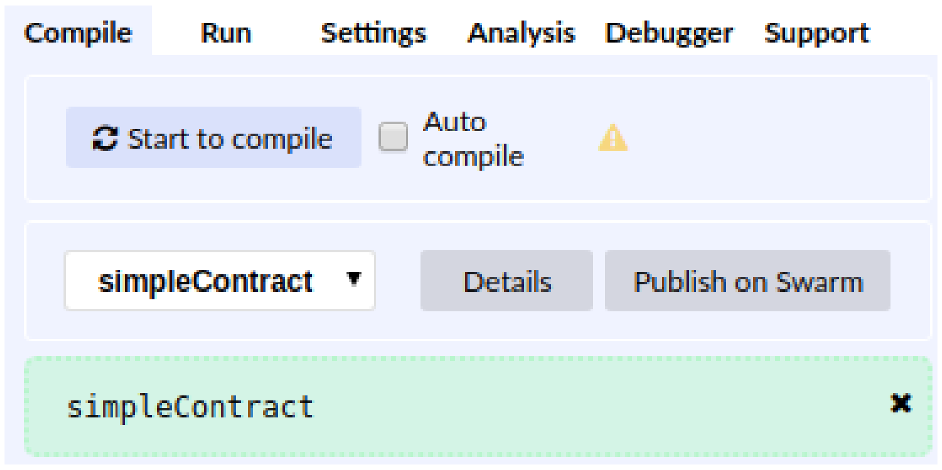


Figure 14: Remix IDE Online: Compile Options

Then click the **Run** tab on the right menu. we connect IDE to the node, deploy the contract and get the deployment address from this page.

Firstly, we change the Environment as Web3 Provider from 1 . It will pop-up a new window where we set endpoint of the web3 provider. Change it as **http://localhost:8000**. You can change the account which we execute calls from 2. Also, It will show the balances of the accounts. If you have enough ether in the wallet, you can click 3 to deploy the contract. The deployment will take some time. Then, you need to copy and store the address from 4. It is the deployment address of contract.

Note : Before deployment you must authorize the account to make transactions. To achieve this, type the following command from geth console :
personal.unlockAccount(eth.coinbase,"PASSWORD",1000)

Now, we will create a new-flow on Node-RED to test after you successfully completed installations(geth, web3 and dependencies) and deployed the smart contract.

As it is seen above, Node-RED flow of this part will consist of six nodes. You have already understood how Debug nodes work, therefore it is not included in this document.

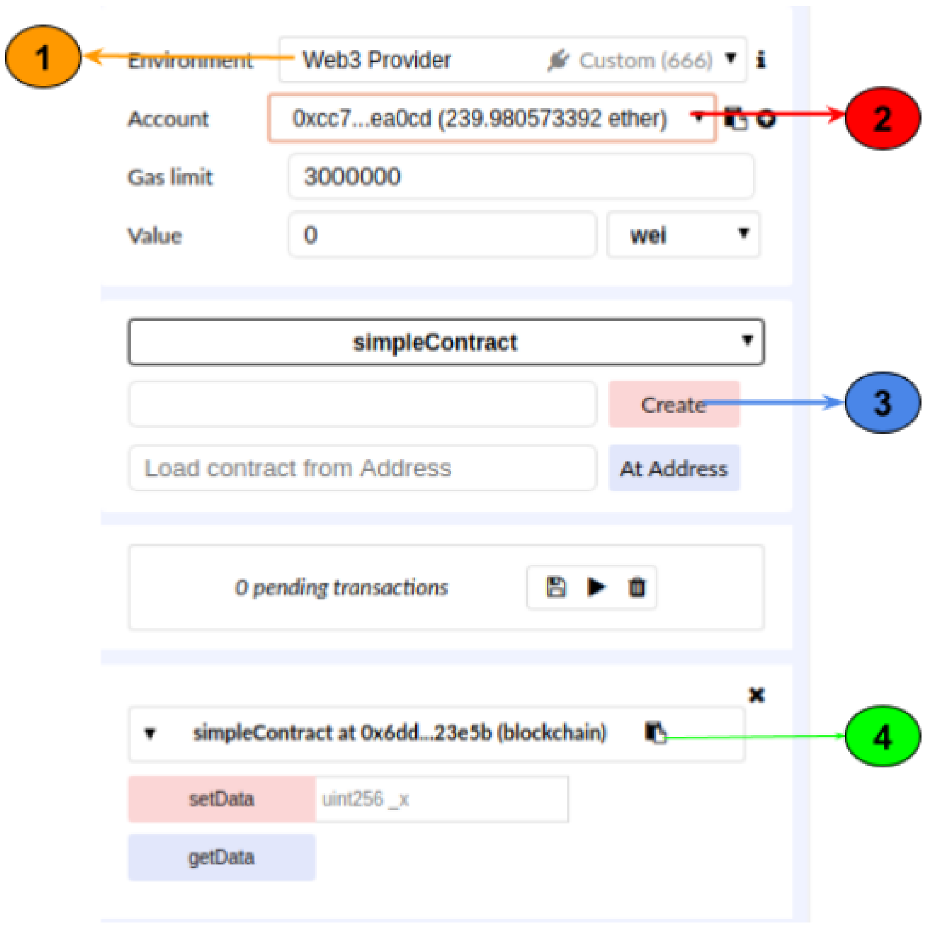


Figure 15: Remix IDE Online: Run Options

- **Timestamp :** Drag-and-drop 'inject' node. It will display as "Time". If it does not display like that, then double-click the node and drop-down "payload" menu to find the timestamp. You need to set "repeat" interval as every 5 seconds. Do not forget to enable 'Inject once after 0.1 seconds'.
- **Getter :** Drag-and-drop 'function' node. Set its name as Getter. The function details are given below. This function simply provides connection to web3 provider and lets you make contract calls. It will call 'getData' function from smart contract that you implemented above.

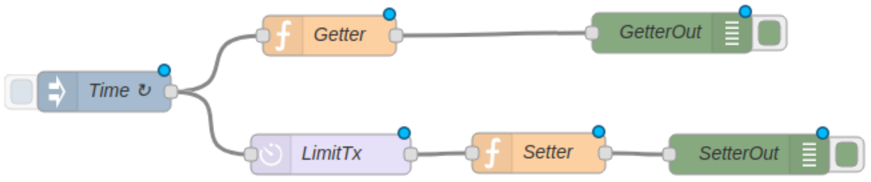


Figure 16: First Flow for interaction with Ethereum

When the flow is deployed, you must read ‘x‘ value from smart contract. To achieve this, we append a ‘Debug‘ node which gets the output of this function as input.

```
// This part must be placed at the beginning of each
    function which you interact with smart contract.
/*########### STARTS HERE ~ WEB3 PART ##########*/
var Web3 = global.get('web3');
var abi = _______ // the ABI goes here
var main_account = _______  //the coinbase address;
var  web3;
//web3.js, checking if node is running
web3 = new Web3(new Web3.providers.HttpProvider("http://
    localhost:8000"));
/*########### ENDS HERE ~ WEB3 PART ##########*/
// Check whether you connected to web3provider. Otherwise,
    it changes the message of payload
if (!web3.isConnected()) {
    msg.payload = "node not connected";
}
else{
  var simpleContract = web3.eth.contract(abi).at("_______")
      ; // deployed contract address
  var password = "_______"; //the password
  web3.personal.unlockAccount(main_account,password,30);
  var x = simpleContract.getData({from:main_account});
  msg.payload=x;
}
return msg;
```

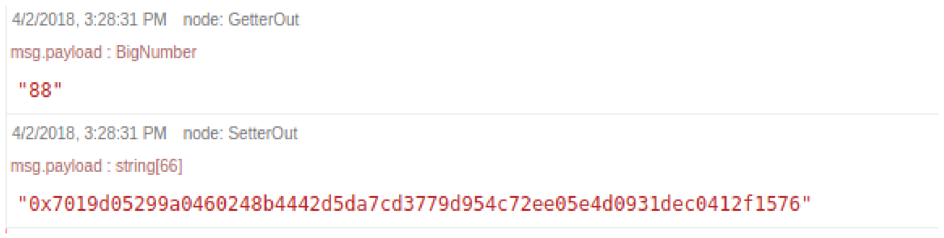
- **LimitTx :** Since the transaction throughput on Ethereum Network is limited by Gas Amount per block, we will set a boundary on transactions. Check that ‘Delay‘’ node’s parameters are set as ‘Rate Limit‘

for ‘all messages‘ as ‘1 message per 60 seconds‘. Also, you don’t need intermediate messages, so click ‘Drop Intermediate Messages‘.

- **Setter :** Drag-and-drop ’function’ node. Set its name as Setter. It will call the ‘setData‘ function from smart contract that you implemented above. When the flow is deployed, we achieve to complete a transaction that changes the value of stored variable in smart contract. we again append a ‘Debug‘ node which gets the output of this function as input. The output of this function will be the hash of transaction that the node pushed to network.

```
/* WEB3 PART SHOULD PLACED AT THE BEGINNING*/
var simpleContract = web3.eth.contract(abi).at("_______");
    // deployed contract address
var password = "_______"; // the password
web3.personal.unlockAccount(main_account,password,30);
var retcont = simpleContract.setData(msg.payload%100,{from:
    main_account});
msg.payload=retcont;

return msg;
```



```
4/2/2018, 3:28:31 PM   node: GetterOut
msg.payload : BigNumber
 "88"
4/2/2018, 3:28:31 PM   node: SetterOut
msg.payload : string[66]
 "0x7019d05299a0460248b4442d5da7cd3779d954c72ee05e4d0931dec0412f1576"
```

Figure 17: Output of First Flow : Getter and Setter functions

4 Sending Sensor Data as transaction

In this part of experiment, we will implement a smart contract to store sensor readings in Ethereum network. the requirements are as follows:

- Contract should store name and geolocation as strings
- Define a new struct that stores LDR, Microphone readings and address of the sender
- Create a constructor that takes name and geolocation as parameters
- Write a function that adds sensor data to *mapping* structure
- Write a function that returns all timestamps of data samples
- Write a function that returns readings at a specific timestamp

```
pragma solidity ^0.4.18;
contract deviceMonitor{
    string name; // name of device
    string geoloc; // geolocation of device

    struct sensorData{
        uint ldr; // reading from ldr sensor
        uint mic; // reading from microphone module
    }

    mapping(uint => sensorData) dataCollection; // map from
       timestamps to corresponding readings
    uint [] timestamps ;

    function updateSensorData(uint _timestamp, uint _ldr, uint
       _mic) public{
        dataCollection[_timestamp].ldr=_ldr;
        _________________ /* Set microphone Value */;
        dataCollection[_timestamp].deviceAddr =msg.sender;
        timestamps.push(_timestamp);
    }
    function deviceMonitor(string _name,string _geoloc) public{
        name=_name;
       _________________ /* Set geolocation from given parameter
          */
    }
    function getTimeStamps() public view returns(uint []){
        return timestamps;
    }
    function getDataAtTime(uint _timestamp) public view returns(
        string,string,uint,uint){
        /* Function that returns name&geolocation of device,
            with readings corresponding to _timestamp */
     _________________    //Hint : Return multiple values  by
         using return(val1,val2,val3)
    }
}
```

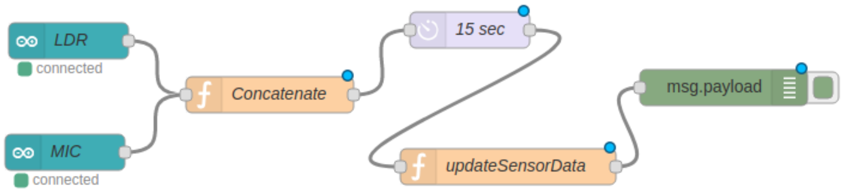


Figure 18: Second Node-Red Flow

As it is seen above, Node-RED flow of this part will consist of seven nodes. You have already understood how Debug nodes work, therefore it is not included in this document.

- **15 Sec :** Check that ‘Delay‘ node’s parameters are set as ‘Rate Limit‘ for ‘all messages‘ as ‘1 message per 15 seconds‘. Also, you do not need intermediate messages, so click ‘Drop Intermediate Messages‘.

- **Concatenate :** Drag-and-drop ’function’ node. Set its name as Concatenate. It will combines readings from two different sensors. It is similar to node in first Hands-On-Session.

```
context.ldr = context.ldr || 0.0;
context.mic = context.mic || 0;
if (msg.topic === '_______') { // For Analog Readings
    append A before pin number such as A1,A2
  context.ldr = msg.payload;
} else if (msg.topic === '_______') {
  context.mic = msg.payload;
}
msg = {
    payload: {
        "LightSensorVal" : context.ldr,
        "MICSensorVal" : context.mic,
        "timeStamp" : Date.now()
    }
};
return msg;
```

Listing 3: Body of concatenate function

- **updateSensorData :** Drag-and-drop 'function' node. Set its name as updateSensorData. It will call the 'updateSensorData' function from smart contract that you implemented above.

```
var Web3 = global.get('web3');
var abi = _______ ; // smart contract ABI GOES HERE ;
var main_account = "_______";// the ACCOUNT;

var  web3;
//web3.js, checking if node is running
web3 = new Web3(new Web3.providers.HttpProvider("http://
    localhost:8000"));
if (!web3.isConnected()) {
    msg.payload = "node not connected";
    return msg;
}
var deviceMonitor = web3.eth.contract(abi).at("_______");
    // deployed contract address
var password = "______"; // the password
web3.personal.unlockAccount(main_account,password,30);

____________________ // You need to pass parameters to
    smart contract
//Hint : msg.payload.LightSensorVal ....
var retcont = deviceMonitor.updateSensorData(______________
    ); // Parameters
msg.payload=retcont;

return msg;
```

Listing 4: updateSensorData

When the flow is deployed, we achieve to complete a transaction that pushes sensor readings to smart contract.

5 Show Sensor Readings

In this part, we create a page that shows the sensor readings which are stored in smart contract. To achieve it:

- Insert an 'HTTP input' node to the flow. Set method as 'GET' and set a 'URL' to make a request.
- Create a new function that pulls timestamps from contract, then pulls sensor readings corresponding to them.
- Render the object in HTML by using 'template' node.
- Attach output of template to 'HTTP response' node.

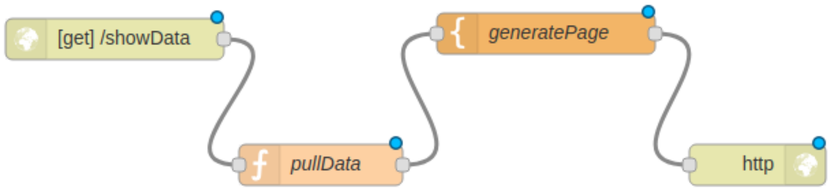


Figure 19: Third Node-RED Flow

pullData : In this function, you need to replace missing parts. Firstly, you have to change the contract address. Then, we call getTimeStamps function from smart contract.

```
// PLACE THE WEB3 PART
var deviceMonitor = web3.eth.contract(abi).at("_______"); //
   deployed contract address
var password = "_____";
web3.personal.unlockAccount(main_account,password,30);
// Get List of TimeStamps from smart contract
var timeStamps = __________; // Write Function
var dataList = []
i = 0;
timeStamps.forEach(function(element) {
    try {
      // Get sensor readings corresponding each timestamp
```

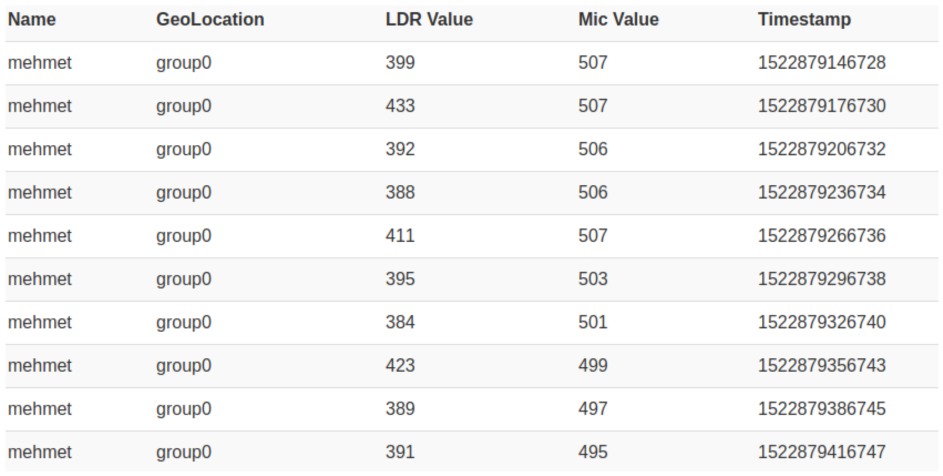
```
        // Hint : Use element.toNumber() while calling the
          function
        var data = _________;
        data.push(element); // appends time stamp to end of data
        dataList.push(data); // appends data to the list
    }
    catch(e) {
        msg.payload = "error";
        return msg;
    }
});
msg.payload = dataList;
return msg;
```

Listing 5: pullData

generatePage : This template generates the response page given below.

```
    <title>IoT Hands On Session 4</title>
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com
       /bootstrap/3.3.7/css/bootstrap.min.css">
    <script src="https://ajax.googleapis.com/ajax/libs/jquery
       /3.3.1/jquery.min.js"></script>
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap
       /3.3.7/js/bootstrap.min.js"></script>
    <table class="table table-striped">
    <tr><th>Name</th><th>GeoLocation</th><th>LDR Value</th><th>
       Mic Value</th><th>Timestamp</th></tr>
    {{#payload}}
        <tr class="">
            <td>{{0}}</td>
            <td>{{1}}</td>
            <td>{{2}}</td>
            <td>{{3}}</td>
            <td>{{4}}</td>
        </tr>
    {{/payload}}
</table>
```

Listing 6: Code of page Generate



Name	GeoLocation	LDR Value	Mic Value	Timestamp
mehmet	group0	399	507	1522879146728
mehmet	group0	433	507	1522879176730
mehmet	group0	392	506	1522879206732
mehmet	group0	388	506	1522879236734
mehmet	group0	411	507	1522879266736
mehmet	group0	395	503	1522879296738
mehmet	group0	384	501	1522879326740
mehmet	group0	423	499	1522879356743
mehmet	group0	389	497	1522879386745
mehmet	group0	391	495	1522879416747

Figure 20: A simple Collection of Sensor Readings

6 Acknowledgement

This project has been supported by Scientific Research Fund of Bogazici University under grant no: 13500

References

[1] Vala Afshar. Cisco: Enterprises are leading the internet of things innovation.

[2] Ethereum Community. A next-generation smart contract and decentralized application platform, 2014.

[3] Ameer Rosic. Ethereum Developer: How to Become One, 2018. https://blockgeeks.com/guides/ethereum-developer/.

[4] Swarm Community. What is Swarm ?, 2017. Available at `https://github.com/ethersphere/swarm`.

[5] Felix Lange. Geth, 2017. https://github.com/ethereum/go-ethereum/wiki/geth.

[6] Fabian Vogelsteller. Geth, 2017. https://web3js.readthedocs.io/en/1.0/.