

# RealMeh? (Decentralised Newspaper)

Antonio Miguel Canlas Quizon 1003014

## 1. Introduction

Given the COVID-19 situation, there was a lot of fake and real news being disseminated. And I believe the ignorance of these issues implicated into the worsening of the spread of the virus. When the situation was worsening in China, numerous websites and blogs were spreading the news that the virus was not lethal, and would not spread internationally. Hence, this resulted in countries being oblivious of such threats.

Fake news can come in different levels including but not limited to comedy, clickbait, misleading, and inaccurate/unreliable news. As a result, it is vital to combat this issue especially in this modern technological world whereby information can easily be spread through the Internet. We need a platform that would provide masses with credible news, as well as prepare them in determining illegitimate news articles. As you can see from Figure 1, there were 70,000 shares for that specific news. This shows the heavy implications and influence of fake news to the masses.

Therefore, I implemented an anti-fake news distributed application. In essence, the application would allow people to submit news headlines, and users can pay (gas+fee) to vote whether it is real or fake. In order to bring motivation, users (broadcasters/journalists) who submit headlines that were voted real will receive a small reward in terms of cryptocurrency(ether). This would also prepare the general public in determining real and fake news in the future.



Figure 1: FAKE NEWS Example

<b>1. Introduction</b>	<b>1</b>
<b>2. DApp Overview</b>	<b>3</b>
2.1 Why is Blockchain applicable?	3
2.2 Architecture	6
<b>3. Distributed Application (Blockchain)</b>	<b>10</b>
3.1 Front-End Description	10
3.1 Client-Side Use Cases	11
3.1.1 Submit News for a poll	11
3.1.2 Vote for Real/Fake news	13
3.1.3 View real news	14
3.1.4 Claim Reward if News is real	16
3.2 Back-end Description (source code will be provided)	17
3.2 Contract-Side Use Cases	17
3.2.1 Sending the reward (for real news)	17
3.2.2 Tallying the votes and 3.2.3 Ending Poll for current news	18
3.2.4 Storing the News in the Blockchain	18
3.2.5 Allowing users to view more details about the news articles	19
<b>4. Overview (UML Diagrams)</b>	<b>20</b>
4.1 Use Case Diagram	20
4.2 State Transition Diagram (Voting)	20
4.3 Sequence Diagram	21
4.4 Flow Diagram	22
<b>5. Extensive Testing</b>	<b>23</b>
5.1 Good Use Case Tests	23
5.2 Error/Failure Use Case Tests	25
<b>6. Analysis</b>	<b>26</b>
6.1 Security	26
6.2 Performance (Gas Usage Analysis)	28
<b>7. Future Development</b>	<b>29</b>
7.1 Gas Improvement	29
7.2 Vulnerabilities	29
7.3 Separate Contracts	30
7.4 Frontend	30
<b>8. Conclusion</b>	<b>30</b>
<b>Instruction Manual</b>	<b>31</b>

## 2. DApp Overview

### Description

The Distributed Application will administer the platform for the people to vote for and view legitimate current issues. It will also enable publishers/journalists/reporters to broadcast headlines that can be determined by the audience for its authenticity. Hence, the DApp would be regarded as the newspaper of the future. In this case, the people have the rights to vote for legitimate news articles, and readers would be able to view the voted counts which will help them create an informed decision. Readers must pay a certain fee (similar to a subscription fee) in order to view and get more details regarding a specific headline.

### 2.1 Why is Blockchain applicable?

#### 2.1.1 Immutability

Once the news article is voted as real, it will be added into the Blockchain and this cannot be altered by hackers/users. If someone wants to corrupt the network, he/she would have to alter every data stored on every node in the network. There could be millions and millions of people, where everyone has the same copy of the ledger. Accessing and hacking millions of computers is next to impossible and costly. Therefore, the results of voting and the respective news articles are recorded and stored securely in the blockchain.

```
9      struct News{
10         string headline;
11         string body;
12         address publisher_address;
13         uint256 real_votes;
14         uint256 fake_votes;
15     }
```

Figure 2: News Struct stored in the Blockchain (Immutable)

#### 2.1.2 Decentralised Technology

Instead of a governing authority, a collection of nodes are responsible for the system's operations. Therefore, users can conveniently store data (news) and vote for them. This provides authenticity, user control and less prone to breakdown as there is no single point of failure.

#### 2.1.3 Availability

As a distributed application, this means that system has a high probability of being operational at a given time. Therefore, this prevents denial of services attacks and ensures the system is operating even at stressful times i.e. COVID-19.

#### 2.1.4 Transparency

Voting and its processes are completely transparent. This ensures there is no form of cheating or biased voting. In addition, everyone will have access to a ‘public ledger’ whereby users can verify past voted news. The vote-counting process is fixed, rules are well established, known to voters and withstands public scrutiny.

```

115     function Vote(bool _choice) public payable NotPublisher {
116         require(msg.value >= 0.01 ether);
117         require(bytes(current_headline).length!=0);
118         if(_choice){
119             metadata[indexOfNews].real_votes += 1;
120         }
121         else{
122             metadata[indexOfNews].fake_votes += 1;
123         }
124         voter_balance[msg.sender] += 1;
125         totalVotes = metadata[indexOfNews].real_votes + metadata[indexOfNews].fake_votes;
126
127         // ensure that this guy has voted.
128         emit Voted(msg.sender,true);
129
130         if(totalVotes>=2){
131             reset_poll();
132         }
133
134     }

```

Figure 3: Transparency of the Voting Process

### 2.1.5 Authenticity

This is in relation to Immutability, the Blockchain’s hashing provides the authenticity for our news data. This allows the broadcasting platform to remain credible.

TX HASH	
0x718fe879472b4f56168ac8836bf34eb6cdf96a14b188b8748ea372888c113b49	
FROM ADDRESS	TO CONTRACT ADDRESS
0xed1a7fa45acC7C9DDdc043FA877f6dD355fD4c48	0xE4c05533578C43Dfd21207413c2d9E9791602488

Figure 4: Authenticity of TXN in Ganache

### 2.1.6 Smart Contracts

Firstly, the transactions (people have to pay to submit/vote) will ensure that they do not carelessly vote/publish news articles as they have a stake on it. Therefore, the smart contracts provide automation of performing credible transactions without third parties involved. In this case, they provide the platform for voting and ensure that reliable publishers are rewarded for their deeds.

### 2.1.7 Tokenization

Our DApp converts ether into two tokens. Firstly, whenever they Vote, they get 1 **Voter Token**. This token can then be used to view more details about the news. Viewing headlines are completely free. On the other hand, if they want to know more about this specific headline, they would need to pay with one Voter Token. Hence, as long as they vote, they would be able to obtain Voter Tokens. However, voting is not entirely free and has a voting fee (with gas included).

```
22      mapping(address=>uint256) voter_balance;

97      function getSpecificNewsDetails(uint256 _index) public voterHasBalance returns(string,uint256,uint256) {
98          voter_balance[msg.sender] -= 1;
99          emit Details(metadata[_index].body,metadata[_index].real_votes,metadata[_index].fake_votes,metadata[_index].publisher_address);
100         return(metadata[_index].body,metadata[_index].real_votes,metadata[_index].fake_votes);
101     }
```

Figure 5: Voter Tokenization and how they are spent

Secondly, a Publisher also receives 1 **Publisher token** for every real news article he publishes. He can then spend this, and claim as ether.

```
24      mapping(address=>uint256) publisher_rewards;

121     function claimReward(uint256 _value) public {
122         require(_value<=publisher_rewards[msg.sender]);
123         publisher_rewards[msg.sender] -= _value;
124         msg.sender.transfer(_value*(1 ether));
125     }
126
127 }
```

Figure 6: Publisher Tokenization and how they are spent

### 2.1.8 Cryptography

Through the public/private key protocol, we can achieve forward anonymity for the voters as they vote for legitimate news. This also prevents people from identity theft attacks given they do not lose their wallet's private keys. Hence, people could not vote for other people's sake and only people with certain eligibility can vote.

## 2.2 Architecture

The Architecture of the DApp is finalised. The technologies I use include **Metamask**, **Web3js**, **Web3py**, **Ganache**, and **Flask**.

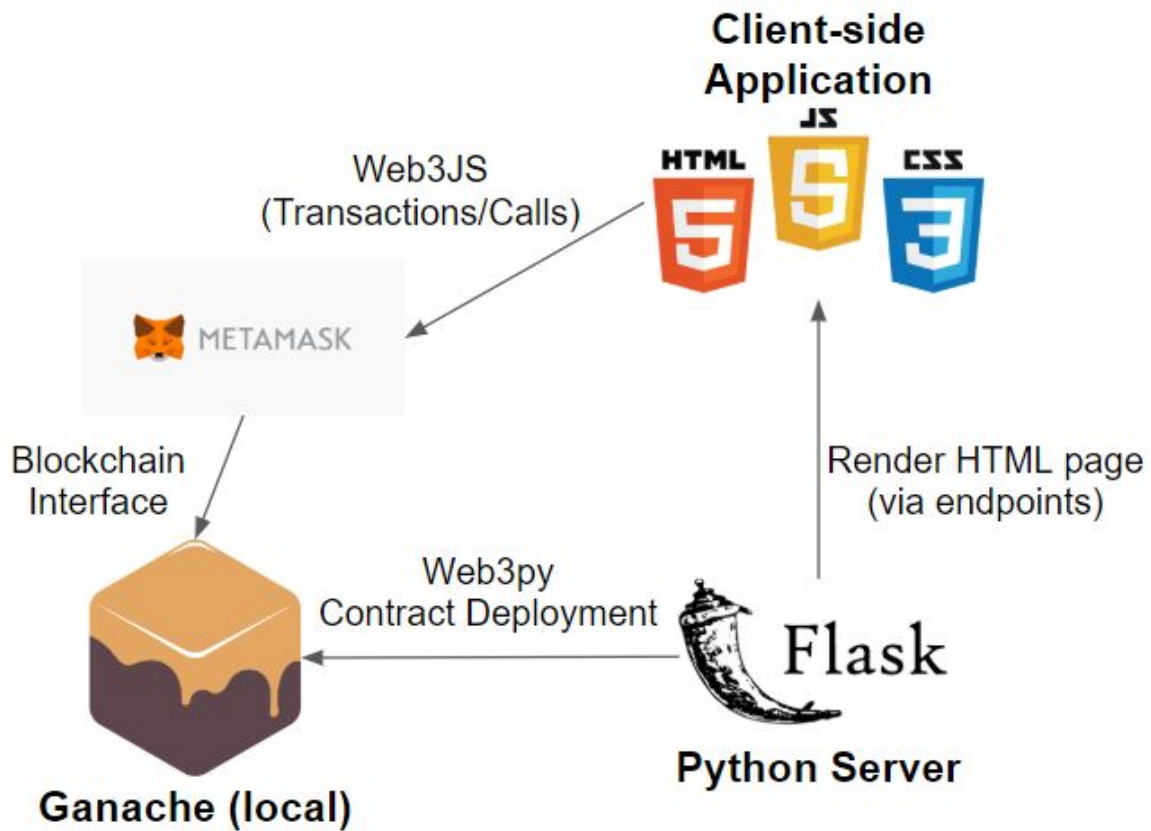
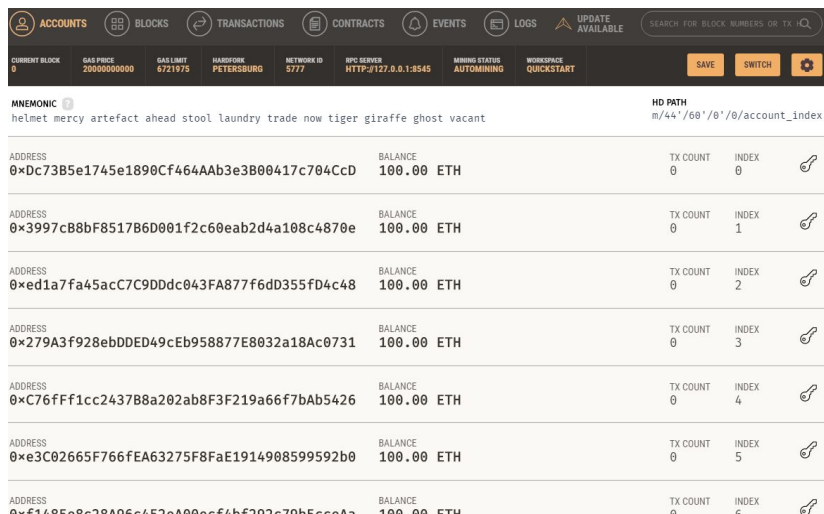


Figure 7: DApp Architecture

## 2.2.1 Ganache

Firstly, Ganache will be my local instance of a Blockchain. As you can see from the Figure below, it serves with a user interface allowing us to visualize the blockchain's activities. From Ganache, we can see the contract creation, transaction calls, transaction receipts, and account balances.



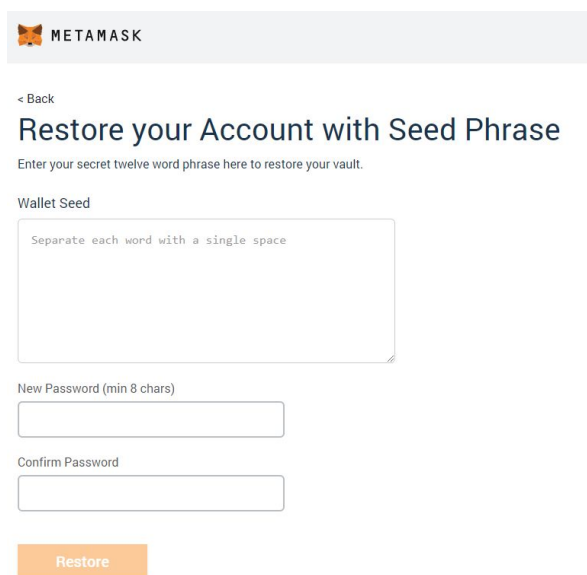
The screenshot shows the Ganache desktop application. At the top, there's a navigation bar with tabs for ACCOUNTS, BLOCKS, TRANSACTIONS, CONTRACTS, EVENTS, LOGS, and UPDATE AVAILABLE. Below this, a status bar displays various network parameters like CURRENT BLOCK, GAS PRICE, GAS LIMIT, HARDFORK, NETWORK ID, RPC ENDPOINT, MINING STATUS, and WORKSPACE. The main area shows a list of accounts with their addresses, balances (100.00 ETH), transaction counts, and indices. A mnemonic phrase is visible at the top left of the main area.

ADDRESS	BALANCE	TX COUNT	INDEX
0xDc73B5e1745e1890Cf464AAb3e3B00417c704CcD	100.00 ETH	0	0
0x3997cB8bF8517B6D001f2c60eab2d4a108c4870e	100.00 ETH	0	1
0xed1a7fa45acC7C9DDdc043FA877f6dD355fD4c48	100.00 ETH	0	2
0x279A3f928ebDDED49cEb958877E8032a18Ac0731	100.00 ETH	0	3
0xC76fFf1cc2437B8a202ab8F3F219a66f7bAb5426	100.00 ETH	0	4
0xe3C02665F766fEA63275F8FaE1914908599592b0	100.00 ETH	0	5
0x5145E8c38A06cF5cA00c6F4bF302c70b5c00A	100.00 ETH	0	6

Figure 8: Ganache (local blockchain)

## 2.2.2 Metamask

The Metamask will be our form of user authentication via the wallets. Using Metamask, we will be able to import the wallets provided in Ganache as seen below. Therefore, we can switch actors through the Metamask User Interface as seen below.



The screenshot shows the Metamask mobile application interface. At the top, there's a header with the Metamask logo and the text 'METAMASK'. Below this, there's a '< Back' link and the title 'Restore your Account with Seed Phrase'. A subtitle reads 'Enter your secret twelve word phrase here to restore your vault.' Below this, there's a 'Wallet Seed' label and a large text input field with a placeholder 'Separate each word with a single space'. Underneath the input field, there are two password fields labeled 'New Password (min 8 chars)' and 'Confirm Password'. At the bottom, there's an orange 'Restore' button.

Figure 9: Tentative DApp Architecture



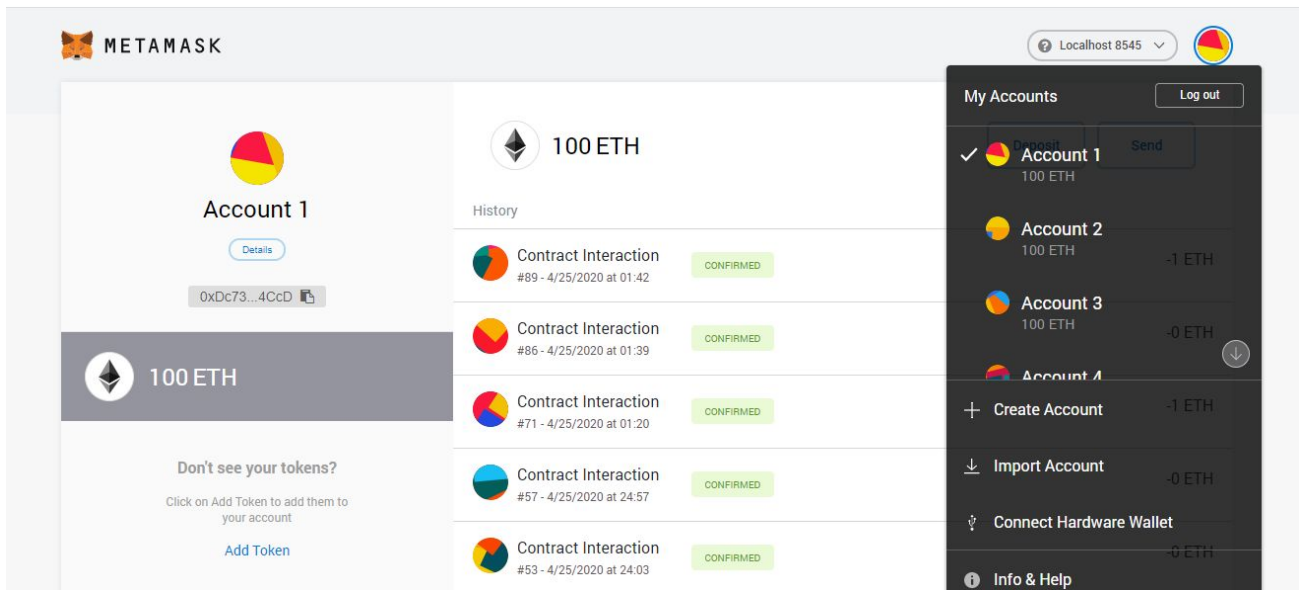


Figure 10: Tentative DApp Architecture

### 2.2.3 Web3py and Flask

Web3py will be our mini-backend that will compile our smart contracts, and automatically deploy to the Blockchain. The Flask framework will then be used to render the HTML files, and inject the required ABI and contract address to the client-side application.

```

13 with open(contract_source_code_file, 'r') as file:
14     contract_source_code = file.read()
15
16 contract_compiled = compile_source(contract_source_code)
17 contract_interface = contract_compiled['<stdin>:Poll']
18 Poll = w3.eth.contract(abi=contract_interface['abi'],
19                        bytecode=contract_interface['bin'])
20
21 # w3.personal.unlockAccount(w3.eth.accounts[0], '') # Not needed with Ganache
22 tx_hash = Poll.constructor().transact({'from': w3.eth.accounts[0]})
23 tx_receipt = w3.eth.waitForTransactionReceipt(tx_hash)
24
25 # Contract Object
26 poll = w3.eth.contract(address=tx_receipt.contractAddress, abi=contract_interface['abi'])

```

Figure 11: Automatically Compile Contract using solc and Web3py

```

39 @app.route('/home')
40 def home():
41     return render_template('home.html', contractAddress = poll.address.lower(), contractABI = json.dumps(contract_interface['abi']))

```

Figure 12: Flask to render an HTML page, and inject contract ABI



## 2.2.4 Web3js and HTML/JavaScript

Web3js will then be used as the library for the Blockchain to interface with our client-side application. The library enables us to send transactions, call our smart contract's functions, and alter the state of the Blockchain from our client-side application. This Web3js library is then imported in our HTML templates which are rendered by the flask server to the client-side application. Our front-end is hence developed using HTML/JavaScript.

```
27     $(document).ready(function() {
28         if (typeof web3 !== 'undefined') {
29             var sc_address = "{{contractAddress}}";
30             var contractABI = web3.eth.contract(JSON.parse('{{contractABI | safe}}'));
31             var contractInstance = contractABI.at(sc_address);
32             contract = contractInstance;
```

Figure 13: Creating our contract instance in the HTML page using **Web3js**

From the web3js, we can start calling and transacting with our contract functions via the contract instance that was passed in from the Flask server. We can see from the figure below, an example of how we interface with the contract using **inline javascript**.

```
171     contractInstance.getCurrentHeadline(function(error, result){
172         if (error) {
173             alert("Please, connect to the network");
174             reject(error);
175         } else {
176             $("#h2#debug").text(result);
177         }
178     });
179
180     $("#a#call_VoteReal").click(function(){
181         contractInstance.Vote(true,{value:web3.toWei(1,'ether')},function(error, result){
182             if (error) {
183                 alert("You are the publisher! or exception");
184                 reject(error);
185             } else {
186             }
187         });
188     });
189
190     $("#a#call_VoteFake").click(function(){
191         contractInstance.Vote(false,{value:web3.toWei(0.2,'ether')},function(error, result){
192             if (error) {
193                 alert("You are the publisher! or exception");
194                 reject(error);
195             } else {
196             }
197         });
198     });
199
200 });
```

Figure 14: Inline Javascript in html templates using **Web3js** to make transactions

### 3. Distributed Application (Blockchain)

The source code will be attached with this report, and can be found at webapp directory.

#### 3.1 Front-End Description

The templates can be found at webapp/static directory.

The client-side application was developed using HTML/CSS/JavaScript. HTML was used to provide the skeleton for the pages. CSS was used to style the pages into modernistic web applications. Lastly, JavaScript (**inline scripting**) was utilised to handle the logic between the user interface and the Web3js library in order to send transactions from the client-side application

Based on the clients' use cases, I started developing different web pages in order to fulfil these requirements. There are 3 different webpages:

- **Submission** - <http://127.0.0.1:5000/submission>
- **Voting/Home** - <http://127.0.0.1:5000/home>
- **View** - <http://127.0.0.1:5000/view>

Therefore, the following use cases can be executed via the corresponding webpage. The front-end can be seen from the following figures in this section.

The Users are identified through their metamask wallets. The only thing that separates them from each other are their wallet addresses. Our web application automatically detects the Metamask wallet address through Web3js, and identifies them as a unique user. Hence, user authentication is handled

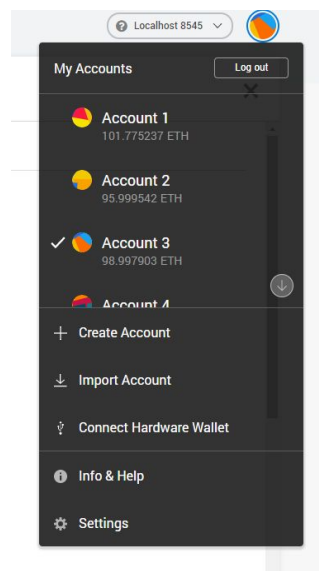
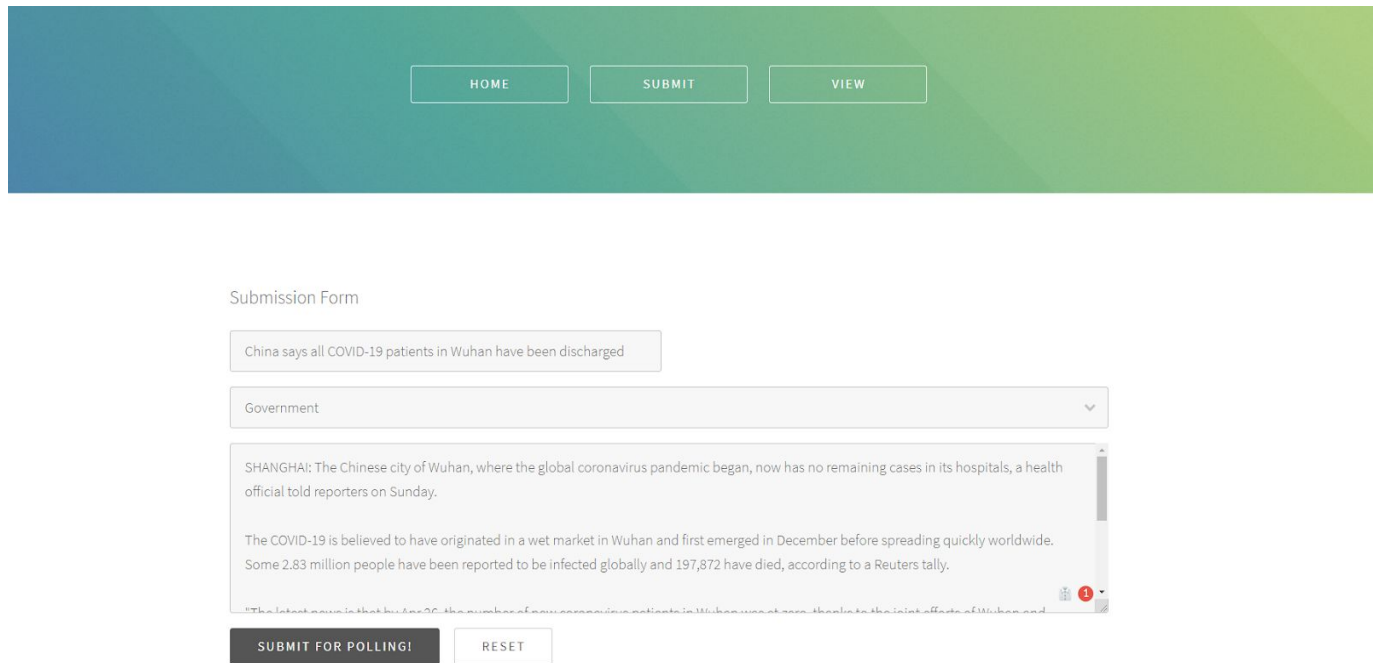


Figure 14: Choosing/Switching Account

## 3.1 Client-Side Use Cases

### 3.1.1 Submit News for a poll

A user can submit his news articles (with headline and body), which will be recorded by the smart contract for polling. This can be done in the **Submission page** as seen below.



The screenshot shows a web interface for submitting news. At the top, there is a navigation bar with three buttons: HOME, SUBMIT, and VIEW. Below this is the 'Submission Form' section. It contains a text input field with the headline 'China says all COVID-19 patients in Wuhan have been discharged'. Below the headline is a dropdown menu currently set to 'Government'. The main body of the form is a large text area containing a news article snippet about Shanghai and the COVID-19 pandemic. At the bottom of the form, there are two buttons: 'SUBMIT FOR POLLING!' and 'RESET'.

Figure 15: Submission Page

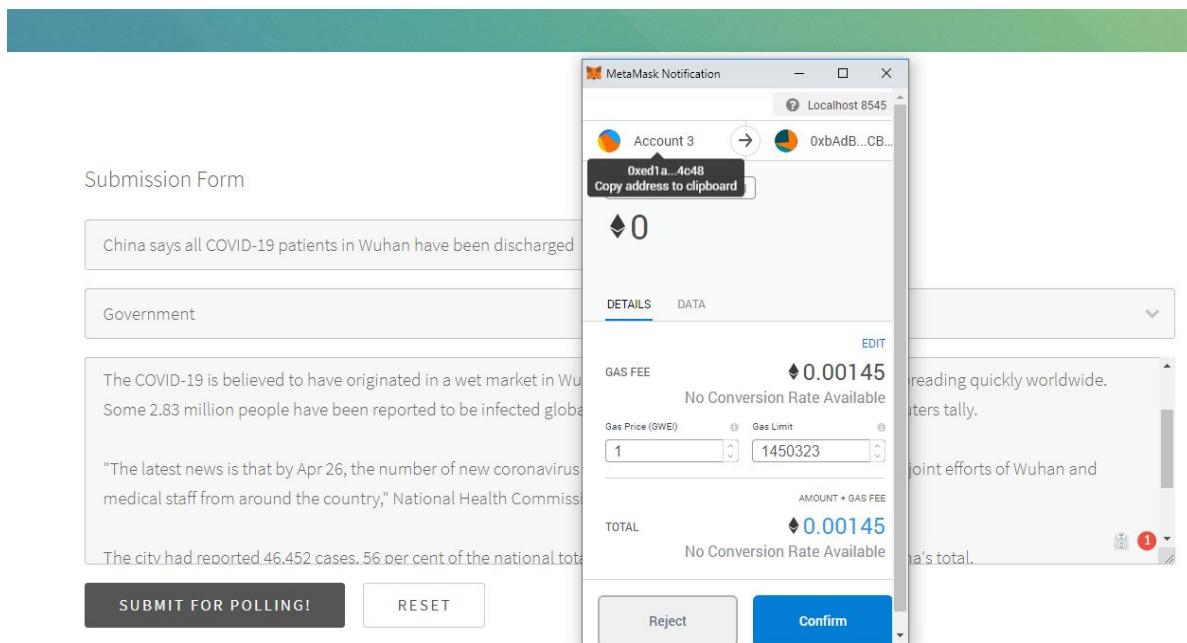


Figure 16: User publishing headline for poll (**Submission Page**)

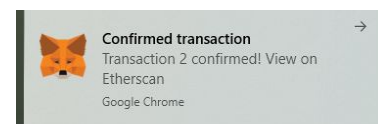
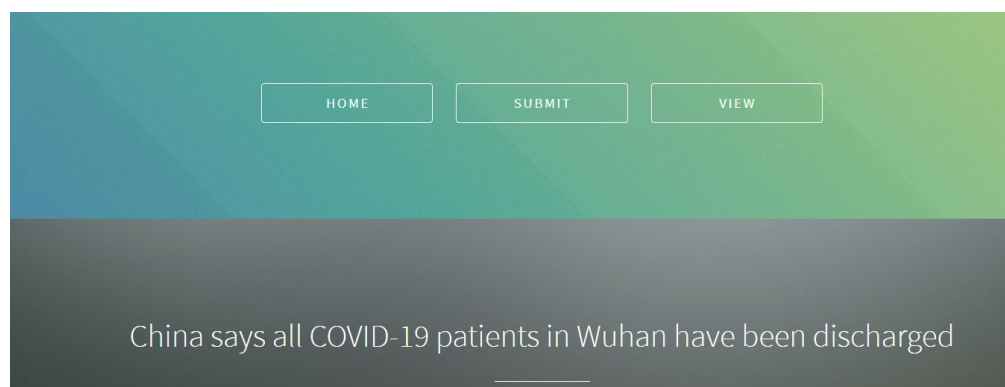


Figure 17: User submitted successfully, redirected to home page with updated headline (**Voting/Home page**)

### 3.1.2 Vote for Real/Fake news

A user can vote for real/fake news on the **Voting page**. He/She must pay the voting fee (including gas) in order to vote successfully. This will increment the voter's balance by 1 token, which will allow her to view other news (another use case).

HOME SUBMIT VIEW

China says all COVID-19 patients in Wuhan have been discharged

REAL!!! FAKE!!!

GET DETAILS REFRESH VOTER BALANCE Voter Balance:

REFRESH PUBLISHER BALANCE Publisher Balance:

CLAIM REWARD Deposit Value



Pay to see details! (uses voter balance)

Total Real Votes: Total Fake Votes:

Adipiscing a commodo ante nunc accumsan interdum mi ante  
adipiscing. A nunc lobortis non nisl amet via volutpat ac lacus nascetur  
ac non. Lorem curae eu ante amet sapien in tempus ac. Adipiscing id  
accumsan adipiscing ipsum.



Figure 18: User submitted successfully, redirected to home page with updated headline (**Home/Voting page**)

### 3.1.3 View real news

A user can see the current headline (currently being voted) at the **Voting page**. The user can also pay for more details such as the body of the news, and the number of real and fake votes. This can be seen from below

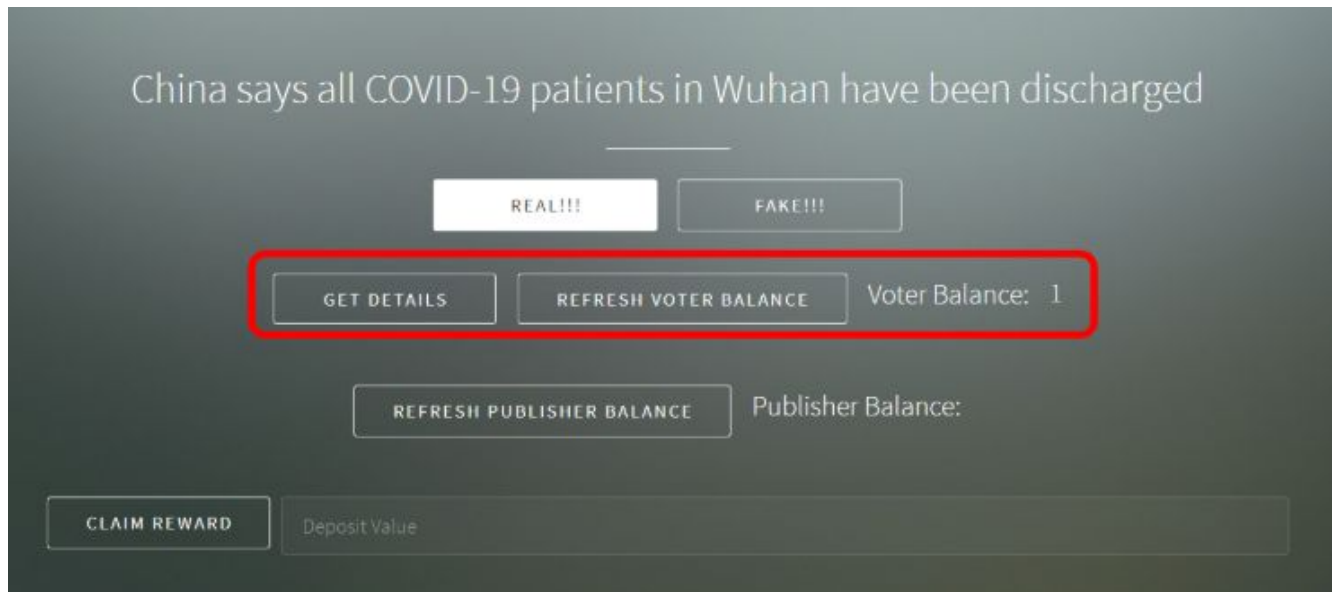


Figure 19: Voter Balance allows to press Get Details of current headline

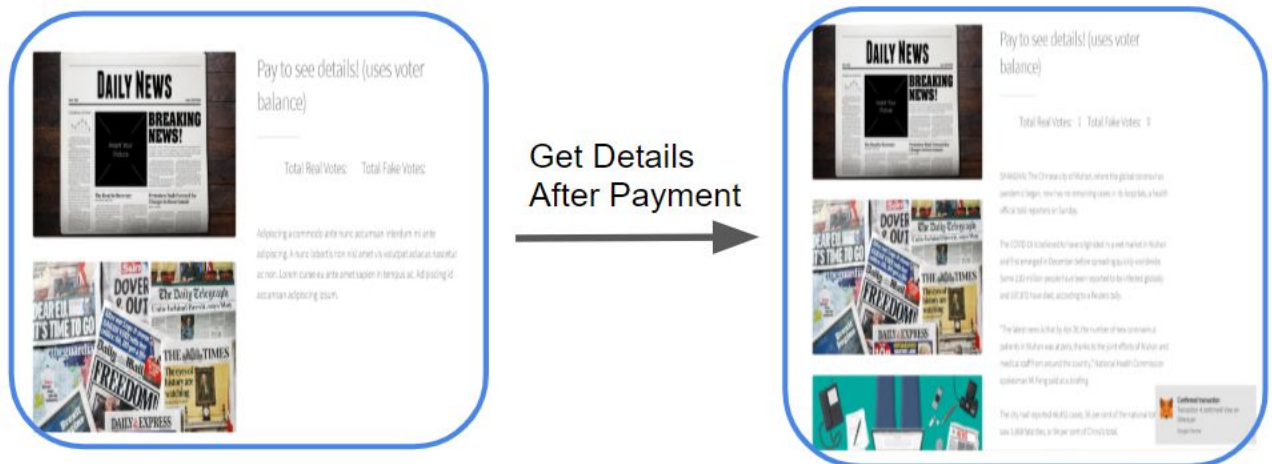


Figure 20: Can now see the body of the news after paying to get more details

Also, the user can view specific headlines, get the total number of news on the blockchain, and get details of a specific headline through the **Viewing** page. The figure below illustrates how the use cases can be performed in the webpage.

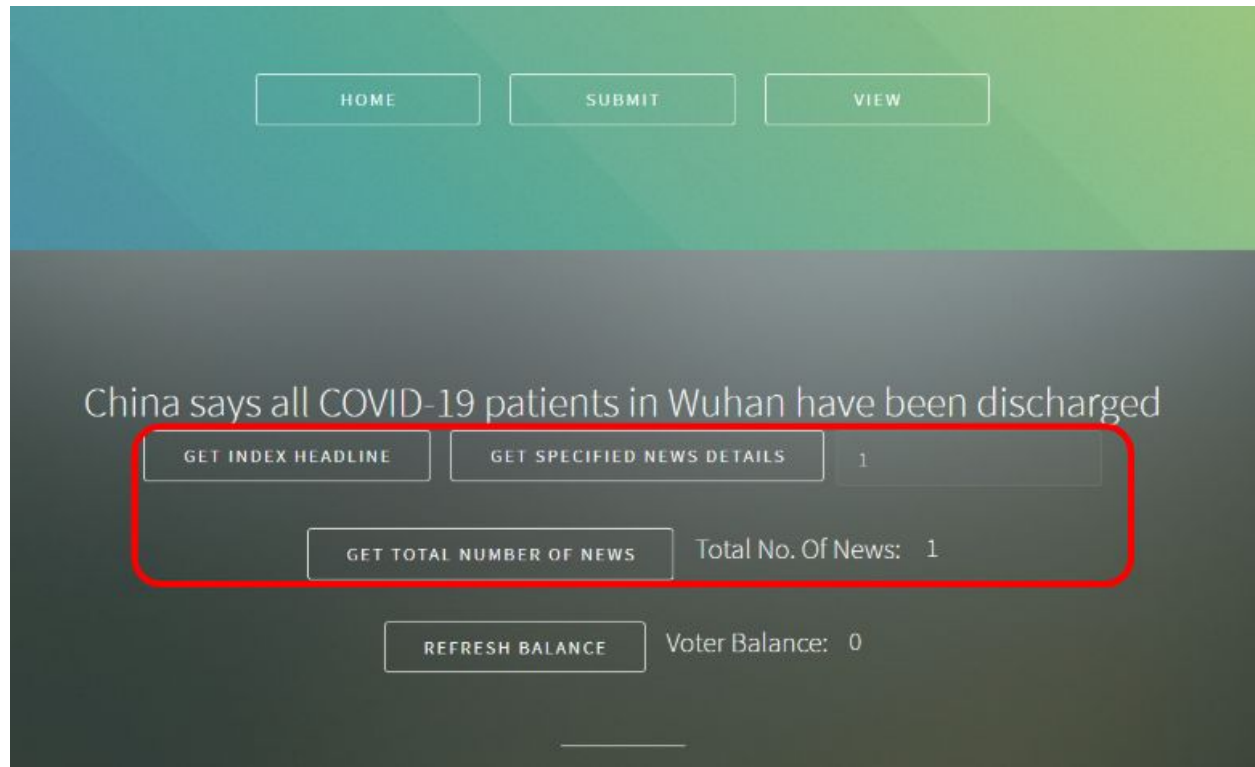


Figure 21: Can View a specific index of headline.

**Limitation:** Solidity does not allow to return arrays of arrays. Hence, it is not possible to return a list of strings such as the headlines and bodies. This will be discussed more later under future development. Therefore, I was not able to implement a table that allows users to view multiple news at the same time. Nevertheless, the functionality of viewing news is fulfilled, and hence there is a good room for improvement. That just means this project has potential!



### 3.1.4 Claim Reward if News is real

If the publisher provides real news, he/she will be rewarded a certain sum of ether. This will automatically be handled and done by the smart contract, once the poll has ended (reached the required number of votes). The publisher will have his publisher balance incremented, and will be able to convert these tokens into the real ether by claiming their reward.

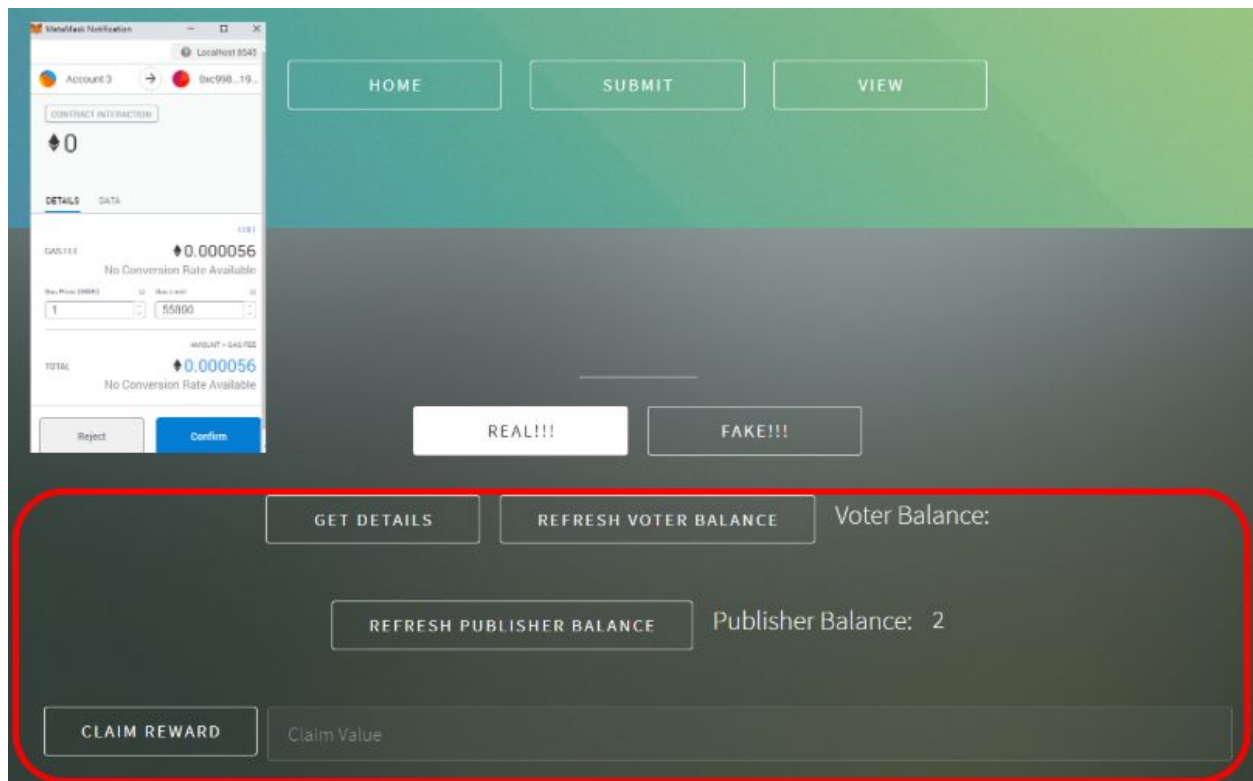


Figure 22: Claim Reward for Publishing Real News (Voting/Home Page)

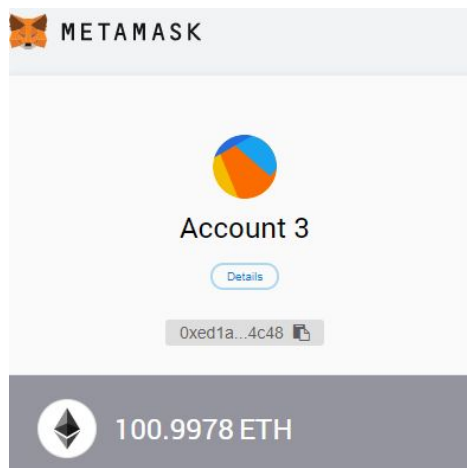


Figure 23: wallet balance after claiming successfully

## 3.2 Back-end Description (source code will be provided)

The flask server is at `webapp/server.py`

The contract is at `webapp/poll.sol`

### 3.2 Contract-Side Use Cases

On the other hand, the use cases for the smart contract consists of several use cases which were implemented in solidity through smart contract functions. Some of these functions are public, payable, and view functions. They are also implemented with certain modifiers depending on the use-case.

#### 3.2.1 Sending the reward (for real news)

This provides rewards for publishers who provide real news to the chain. The publishers are rewarded after the poll is done, hence the `reset_poll()` function automatically increments the publisher's token balance. However, the publisher will need to claim if he/she wants to convert it into the ether. Initially, I transferred the ether automatically, but due to a Denial of Service vulnerability, I decided to have a withdrawal/claim function. This will be elaborated on the security analyses.

```
151 ▼ function reset_poll() internal {  
152 ▼     if((metadata[indexOfNews].real_votes/totalVotes) * 100 >= 60 ){  
153         real_headlines.push(indexOfNews);  
154         // metadata[indexOfNews].publisher_address.transfer(2 ether);  
155         publisher_rewards[metadata[indexOfNews].publisher_address] += 2;
```

Figure 2: Automatically sends reward as publisher token, saves the headlines, and resets the poll

```
121 ▼ function claimReward(uint256 _value) public {  
122     require(_value<=publisher_rewards[msg.sender]);  
123     publisher_rewards[msg.sender] -= _value;  
124     msg.sender.transfer(_value*(1 ether));  
125  
126  
127     }  
128
```

Figure 24: Allows users to claim their publisher token for 1 ether

### 3.2.2 Tallying the votes and 3.2.3 Ending Poll for current news

The smart contract should be able to tally the votes per news issue and reset the poll once the current issue reaches the targeted vote count. This can be seen in the `Vote(bool _choice)` function in the smart contract which takes in an input of whether it is voted to be real/fake.

```
115     function Vote(bool _choice) public payable NotPublisher {
116         require(msg.value >= 0.01 ether);
117         require(bytes(current_headline).length!=0);
118         if(_choice){
119             metadata[indexOfNews].real_votes += 1;
120         }
121         else{
122             metadata[indexOfNews].fake_votes += 1;
123         }
124         voter_balance[msg.sender] += 1;
125         totalVotes = metadata[indexOfNews].real_votes + metadata[indexOfNews].fake_votes;
126
127         // ensure that this guy has voted.
128         emit Voted(msg.sender,true);
129
130         if(totalVotes>=2){
131             reset_poll();
132         }
133     }
134 }
```

Figure 25: Tallying the votes, saving to metadata, and resetting the poll once finished

### 3.2.4 Storing the News in the Blockchain

The news data should be securely stored in the blockchain to prevent hackers from altering it. Also, this allows us to record the history of our news data for future viewing.

```
21     mapping(uint256=>News) metadata;

103     function submitHeadline(string _headline,string _body) public PollIsEmpty{
104         require(bytes(_headline).length != 0 && bytes(_body).length!=0);
105         indexOfNews += 1;
106         current_headline = _headline;
107         metadata[indexOfNews].headline = current_headline;
108         metadata[indexOfNews].body = _body;
109         metadata[indexOfNews].publisher_address = msg.sender;
110         // if(real_headlines.length)
111         // real and fake votes already instantiated to 0
112     }
```

Figure 26: Storing in metadata

### 3.2.5 Allowing users to view more details about the news articles

The contract provides voter tokens that allow users to pay for more details regarding the news articles that are published. This can be seen from below.

```
98 ▼ function getCurrentNewsDetails() public voterHasBalance returns(string,uint256,uint256) {
99     voter_balance[msg.sender] -= 1;
100     emit Details(metadata[indexOfNews].body,metadata[indexOfNews].real_votes,metadata[indexOfNews].fake_votes,metadata[indexOfNews].publisher_address);
101     return(metadata[indexOfNews].body,metadata[indexOfNews].real_votes,metadata[indexOfNews].fake_votes);
102 ▼ }
103
104 function getSpecificNewsDetails(uint256 _index) public voterHasBalance returns(string,uint256,uint256) {
105     voter_balance[msg.sender] -= 1;
106     emit Details(metadata[_index].body,metadata[_index].real_votes,metadata[_index].fake_votes,metadata[_index].publisher_address);
107     return(metadata[_index].body,metadata[_index].real_votes,metadata[_index].fake_votes);
108 }
```

Figure 27: Returning News Details in exchange for 1 voter token

### 3.2.6 Callable View functions

Aside from the public and/or payable functions, there are several callable view functions that can be called from their local nodes to read state variables. For example, `getVoterBalance()` that returns the number of voter tokens for that user. **More details can be found in [webapp/Poll.sol](#).**

```
72     function getVoterBalance() public view returns(uint256){
73         return(voter_balance[msg.sender]);
74     }
```

Figure 28: Example of callable view function

## 4. Overview (UML Diagrams)

By following the Elements of Software Construction, the system's design was generated through Unified Modelling Language diagrams. This allowed visualization of the system's functionalities, states, and use cases. The diagrams are almost identical to the proposal's diagrams as this is the software design that I followed in implementing my DApp.

### 4.1 Use Case Diagram

This is an overall Use Case Diagram between the core actors of the DApp.

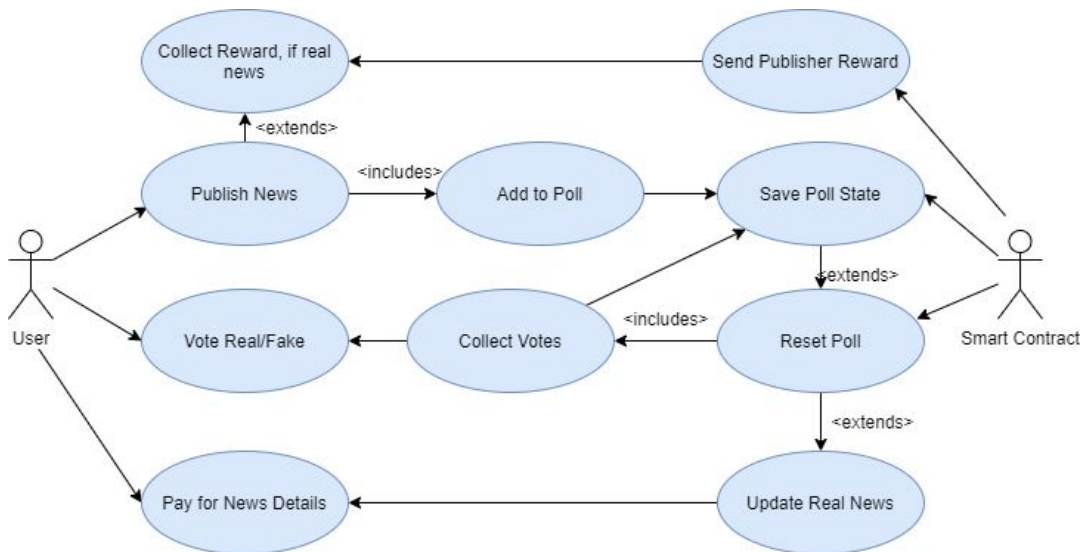


Figure 29: Use Case Diagram between actors

### 4.2 State Transition Diagram (Voting)

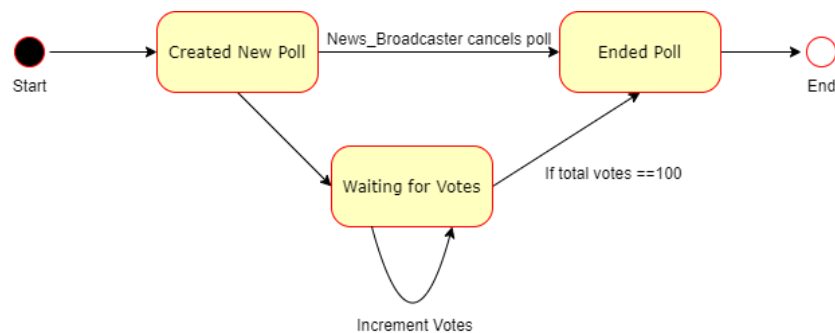


Figure 30: State Diagram for Voting

## 4.3 Sequence Diagram

The diagram below represents the sequence of the system's actions, as real news issues are published in the blockchain. It demonstrates the steps for the submission and voting of real/fake news. It also illustrates the communication between different actors and components on the system.

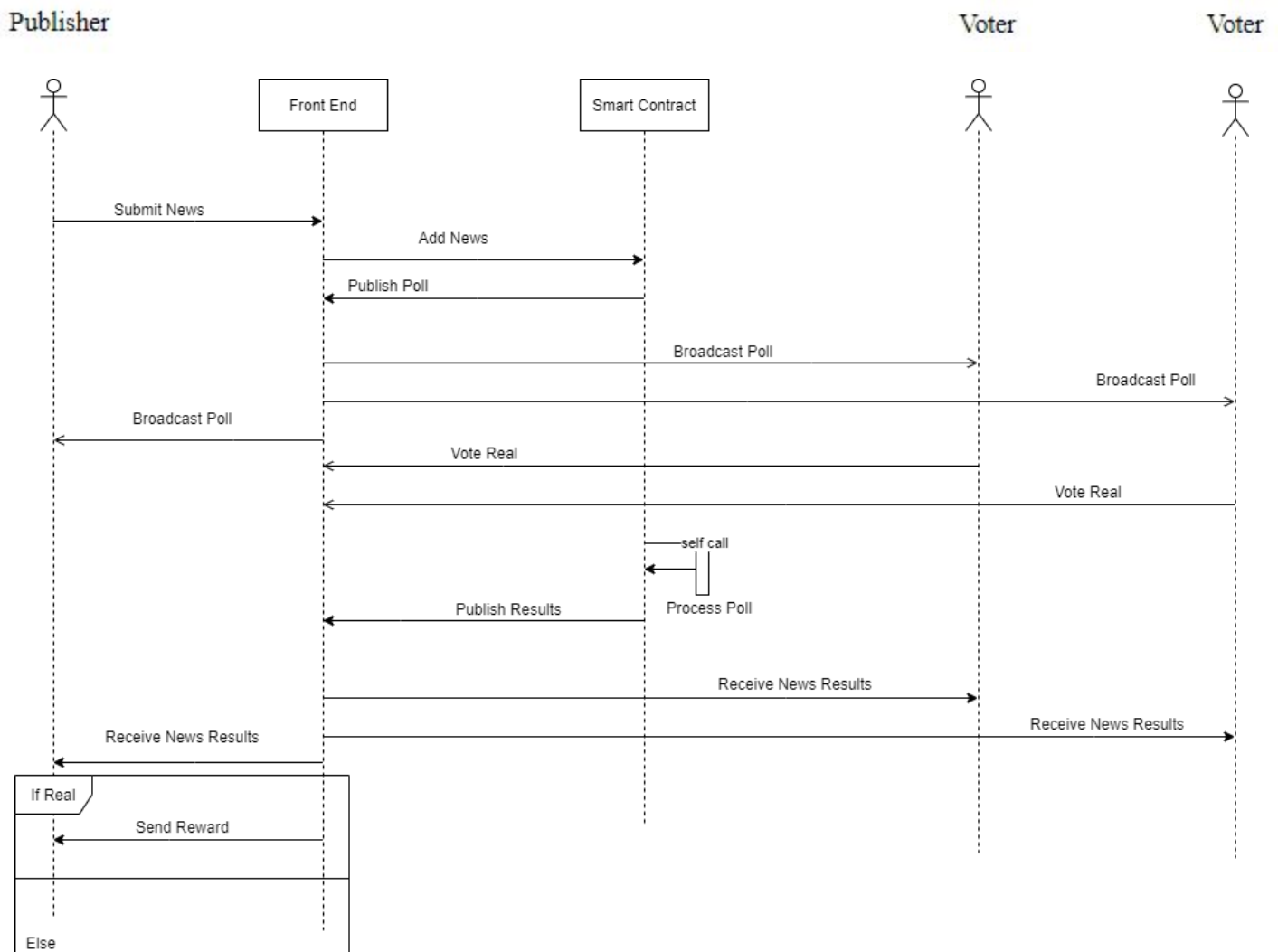


Figure 31: Sequence Diagram of the system's operations

## 4.4 Flow Diagram

This is the finalised flow diagram for the Blockchain's state sequence, as the smart contract is initialized.

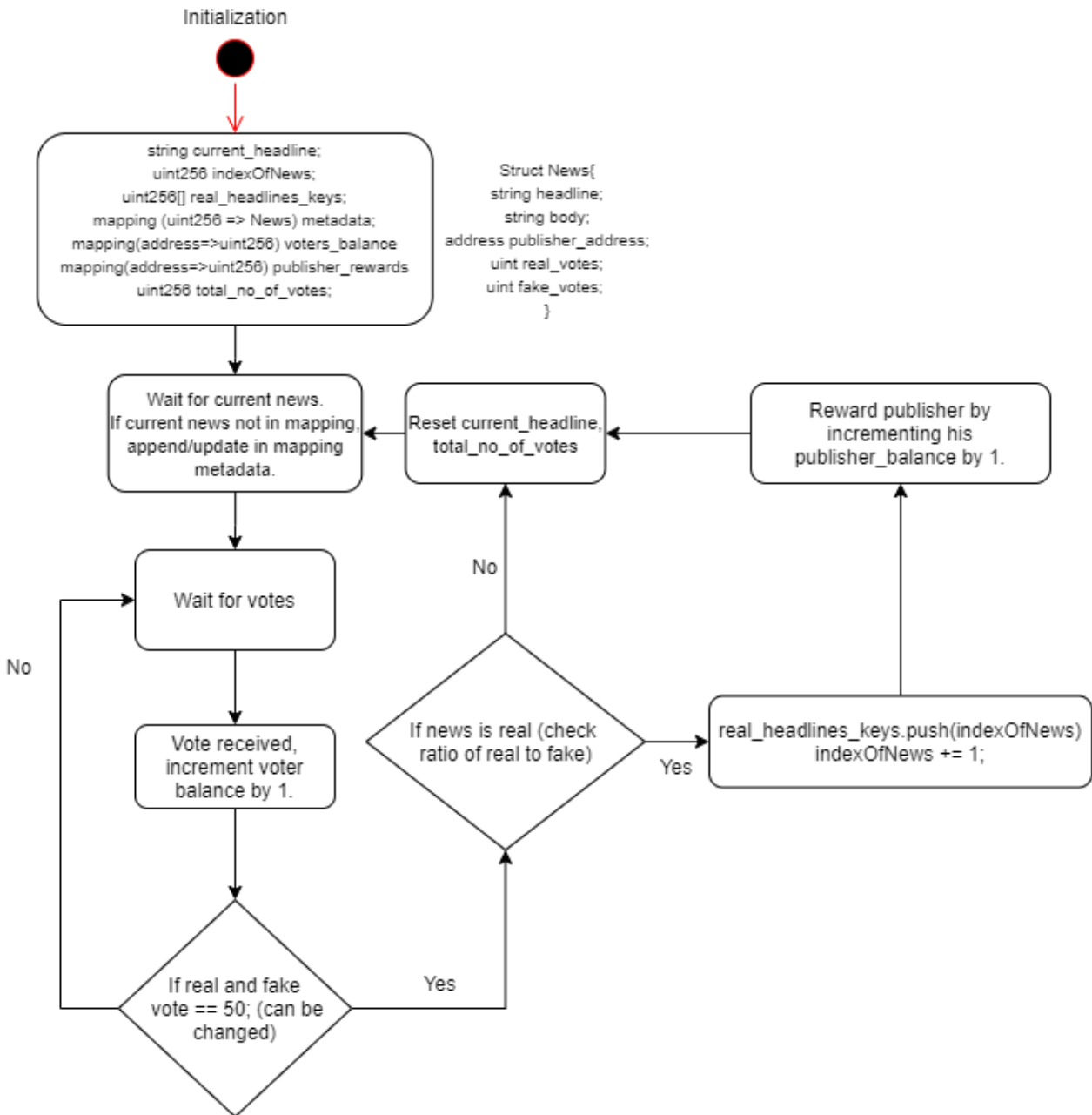


Figure 32: Flowchart of backend runtime



## 5. Extensive Testing

The testing is done through **pytest**, **py-evm**, **web3py**, and **eth-tester**. This allows us to not have any local blockchain and wallets. Therefore, we can test solely on the *Python* compiler. Pytest is responsible for the unit tests, and ensuring assertions are handled for correct and error use cases respectively. Py-evm is used to simulate the Ethereum protocol in Python. Web3py is used to interface the virtual blockchain in the py-evm environment. Lastly, eth-tester provides testing tools such as wallets and transaction methods.

Setting up this testing environment was not easy, as a lot of the dependencies were not compatible with Windows. Thus, a normal `pip install` does not work. I had to manually clone from their respective repositories, build and edit some source codes. Nevertheless! I am proud I have done this, as it ensures that my solidity contract can be extensively tested. There are a total of **20** unit/integration tests, and **~300 lines** of testing to ensure the robustness of the DApplication.

**Refer to `webapp/apptest.py` for test data/source code. Running the tests can also be seen in my video submission.**

### 5.1 Good Use Case Tests

Test Name	Test Description/Steps	Expected Output (Assert)
test_owner	<ul style="list-style-type: none"><li>• Basic test that deploys contract</li><li>• asserts owner</li></ul>	Check if wallet base account is same as output of <code>getOwner()</code>
test_submit_headline_success	<ul style="list-style-type: none"><li>• Deploys contract</li><li>• Submits the headline "asdasd".</li></ul>	Check if <code>getCurrentHeadline()</code> returns 'asdasd'
test_submit_headline_and_check_if_total_news_is_incremented	<ul style="list-style-type: none"><li>• Deploys contract</li><li>• Submits the headline "asdasd".</li></ul>	Assert if total news count == 1
test_successful_vote_with_submitted_headline	<ul style="list-style-type: none"><li>• Deploys contract</li><li>• Submits the headline "asdasd".</li><li>• Another user vote real, with enough value</li></ul>	Assert if Voted event is emitted, <code>event.voted == True</code> , <code>event.voter == eth_account.address</code>
test_successful_vote_and_that_voter_is_rewarded_with_vote_	<ul style="list-style-type: none"><li>• Deploys contract</li><li>• Submits the headline</li></ul>	Assert Voter should have his <code>voter_balance</code> incremented

balance	<ul style="list-style-type: none"> <li>• “asdasd”.</li> <li>• Another user vote real, with enough value</li> </ul>	
test_successful_poll_ending_and_that_publisher_balance_is_credited_because_real_news	<ul style="list-style-type: none"> <li>• Deploys contract</li> <li>• Submits the headline “asdasd”.</li> <li>• Another user vote real, with enough value</li> <li>• Another user vote real, with enough value</li> </ul>	<p>Assert publisher balance is incremented, because he is rewarded. The news was voted real, therefore he must be rewarded.</p> <p>Assert old publisher balance = new publisher balance + 2</p>
test_successful_poll_ending_and_that_publisher_balance_is_NOT_credited_because_fake_news	<ul style="list-style-type: none"> <li>• Deploys contract</li> <li>• Submits the headline “asdasd”.</li> <li>• Another user vote real, with enough value</li> <li>• Another user vote <b>fake</b>, with enough value</li> </ul>	<p>Assert publisher balance is not incremented, because the ratio of real to fake has not met the requirements. Hence it is fake news.</p> <p>Assert old publisher balance = new publisher balance</p>
test_successful_poll_ending_and_that_publisher_balance_credited_and_claim_successful	<ul style="list-style-type: none"> <li>• Deploys contract</li> <li>• Submits the headline “asdasd”.</li> <li>• Another user vote real, with enough value</li> <li>• Another user vote real, with enough value</li> <li>• Publisher claims</li> </ul>	<p>Assert publisher evm_tester wallet has been incremented with 2 ether after claiming</p>
test_successful_poll_ending_and_that_publisher_balance_is_credited_then_submit_new_headline_again	<ul style="list-style-type: none"> <li>• Deploys contract</li> <li>• Submits the headline “asdasd”.</li> <li>• Another user vote real, with enough value</li> <li>• Another user vote real, with enough value</li> <li>• Publisher claims</li> <li>• Publisher submits another headline “asdasd”</li> </ul>	<p>Assert current headline == “asdasd”</p>
test_successful_poll_ending_and_that_publisher_is_rewarded_and_poll_reset_with_empty_headline	<ul style="list-style-type: none"> <li>• Deploys contract</li> <li>• Submits the headline “asdasd”.</li> <li>• Another user vote real,</li> </ul>	<p>Assert Poll is empty or current headline == “”</p>

	<ul style="list-style-type: none"> <li>with enough value</li> <li>• Another user vote real, with enough value</li> <li>• Publisher claims</li> </ul>	
--	--	--

## 5.2 Error/Failure Use Case Tests

Test Name	Test Description	Expected Output (assert)
test_submit_headline_fail_cause_poll_is_ongoing	<ul style="list-style-type: none"> <li>• Deploy contract</li> <li>• Submit Headline</li> <li>• Submit Another headline</li> </ul>	Assert exception because tried to submit while poll is still ongoing
test_submit_headline_fail_cause_empty_params	<ul style="list-style-type: none"> <li>• Deploy Contract</li> <li>• Submit empty headline and body for poll</li> </ul>	Assert exception because cannot submit empty headlines/bodies (modifier)
test_failed_vote_because_poll_is_empty	<ul style="list-style-type: none"> <li>• Deploy Contract</li> <li>• Vote with another user</li> </ul>	Assert exception because poll is empty, hence no news to vote for
test_failed_vote_because_msg_value_insufficient	<ul style="list-style-type: none"> <li>• Deploy Contract</li> <li>• Vote with another user, but without paying (msg.value = 0)</li> </ul>	Assert exception because did not pay enough fees
test_failure_successful_poll_ending_and_that_publisher_balance_is_not_credited_because_fake_and_claim_fail_cause_no_balance	<ul style="list-style-type: none"> <li>• Deploys contract</li> <li>• Submits the headline "asdasd".</li> <li>• Another user vote real, with enough value</li> <li>• Another user vote <b>fake</b>, with enough value</li> <li>• Publisher claims but no balance</li> </ul>	Assert exception, because publisher tries to claim with insufficient publisher balance

## 6. Analysis

In terms of analysis, we analysed our Decentralised Application in two components, Security and Performance (Gas Usage).

### 6.1 Security

In terms of security, I believe the Decentralised Application is safe and secure against malicious attacks. Attacks are mainly generated from transfer of balance. This can be in the form of Denial of Service (DOS) attacks or reentrancy attacks. Hence, we need to analyse the parts of our code where we transfer funds to users.

Initially, we wanted to reward the publishers (if they publish real news) automatically. This was very unsafe towards DOS and re-entrancy attacks. Hence, we wanted to separate the logic in order to make the DApp more secure. The publishers can now claim their publisher tokens as ether instead of being automatically transferred. Also, we had to ensure that the publisher rewards are deducted before sending, to prevent further re-entrancy attacks.

```
151     function reset_poll() internal {
152         if((metadata[indexOfNews].real_votes/totalVotes) * 100 >= 60 ){
153             real_headlines.push(indexOfNews);
154             // metadata[indexOfNews].publisher_address.transfer(2 ether);
155             publisher_rewards[metadata[indexOfNews].publisher_address] += 2;
156         }
```

Figure 33: Publisher rewards incremented at reset\_poll

```
121 ▼     function claimReward(uint256 _value) public {
122         require(_value<=publisher_rewards[msg.sender]);
123         publisher_rewards[msg.sender] -= _value;
124         msg.sender.transfer(_value*(1 ether));
125
126
127     }
```

Figure 34: Deduct first, then send reward during claim. (prevents reentrancy)

Another area of security we can safeguard is handling with the contract's balance. Therefore, the safest way to implement this is to only allow the owner of the contract to handle it. Thus we implemented an `onlyOwner` modifier for the deposit/withdraw functions.

```
33     modifier onlyOwner(){
34         require(msg.sender == news_station);
35         _;
36     }

59     // Owner Function
60     function deposit() public payable onlyOwner{
61
62     }
63
64     function withdraw() public onlyOwner{
65         news_station.transfer(address(this).balance);
66     }
```

Figure 35: Only Owner Functions

## 6.2 Performance (Gas Usage Analysis)

For Gas Usage Analysis, we ran every public, interactable contract function and saved the gas cost. There are two costs for analysis:

1. **Transaction costs** are based on the cost of sending data to the blockchain. There are 4 items which make up the full transaction cost
2. **Execution cost** has everything to do with the costs for storing global variables and the runtime of the method calls.

Transaction cost depends on the size of the compiled contract and the execution cost. Hence, we will be recording each function's transaction and execution cost. We will be ignoring gas prices for now, and will look at the absolute gas values instead. This will be done through a gas profile that is available in *Remix IDE*.

Function Name ( <b>non-view/public/payable</b> )	Execution Cost (gas)	Transaction Cost (gas)
submitHeadline ( <code>'asdasd','asdasd'</code> )	111073	134137
Vote(true)	71367	92831
getCurrentNewsDetails()	25393	31665
getSpecificNewsDetails (index=1)	19876	26340
claimReward(2)	14743	21207

As we can see from the table above, `submitHeadline()` and `Vote()` has the highest gas usages as they require the most computation compared to other function calls. Firstly, the `submitHeadline()` has to save the metadata of the news into the blockchain, and hence require gas. Secondly, `Vote()` needs to increment the number of real votes, fake votes, and total votes. It will also tally the result, and if the limit is exceeded, it will reset the poll internally. Thus, these two functions require more gas as compared to the other non-view functions.

On the other hand, the view functions such as `getCurrentHeadline` or `getVoterBalance` are completely free, and do not incur any gas costs as they are reading from the local node. In addition, in order to save gas, the application does not need to initialize state variables during construction.

## 7. Future Development

Possible development (in order to broaden functionalities and improve usability) includes three components. Optimising the gas, increasing the modularity of the system, and improving the user interface/experience design.

### 7.1 Gas Improvement

First of all, there are certain things that can be improved from the gas usage:

1. Use bytes instead of strings because they consume less gas. Currently the application uses strings to store the news articles' headlines and body information

```
9      struct News{
10          string headline;
11          string body;
12          address publisher_address;
13          uint256 real_votes;
14          uint256 fake_votes;
15      }
```

Figure 36: Only Owner Functions

2. Decreasing the transaction cost, by decreasing the total lines of the code. Currently, the Poll.sol contract has a total of 166 lines. If we reduce this, we can reduce the size of the code and hence decrease the transaction cost
3. An array can be used instead of mapping to iterate over the news articles. Currently, the metadata of the news articles are stored in a mapping.

### 7.2 Vulnerabilities

1. A floating pragma is set - It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code. This can be part of our future improvements.
2. Transferring of balance calls with hardcoded gas - This is discouraged as the gas cost of EVM instructions may change in the future, which could break this contract's assumptions. If this was done to prevent reentrancy attacks. Hence we can consider alternative methods such as the checks-effects-interactions pattern or reentrancy locks instead of using msg.transfer for future development.



## 7.3 Separate Contracts

We can have separate for two different functionalities. Since our DApp has several components - (1) Voting, (2) Tokens and Viewing News Articles , (3) Publishing. We can make our system much more modular if we separate the responsibilities into multi-contracts. This will then involve multi-contract communication.

## 7.4 Frontend

And most definitely, the frontend can be further improved to be much more user-friendly, and minimalistic. I allocated and prioritized my time in order to ensure that the DApp functionality is met, and that it is robust enough for security and data integrity. Therefore, a good future development would be improving the user interface of the decentralised application.

## 8. Conclusion

In a nutshell, I was able to create a platform for users to read news articles, submit news to be polled (real/fake) by readers, voting for real/fake news, and lastly, claim rewards as publishers. This enables the community to mitigate deceiving news, and provide a credible platform for broadcasting current issues. There is a good room for improvement, and I believe that necessary for an application with a good potential.

Overall, this project gave me the opportunity to understand the intuition behind Blockchain, and decentralised applications/systems. I would like to thank the course instructors and assistants for offering this course. Hope you enjoyed it as much as I did!

## Source Files

**The source code can be found in the webapp folder.**

**webapp/server.py is the Flask server with solc compiler**

**The HTML templates with inline javascripts (web3js) are found in webapp/templates**

**The smart contract can be found in webapp/Poll.sol**

**The testing data can be found in webapp/apptest.py**

# Instruction Manual

1. Install all the dependencies and requirements:
  - a. Web3py (<https://web3py.readthedocs.io/en/stable/quickstart.html>) (pip)
  - b. Flask (pip)
  - c. Install solc compiler  
(<https://solidity.readthedocs.io/en/latest/installing-solidity.html>)
  - d. Py-solc (pip)
  - e. Ganache
  - f. Metamask plugin for chrome
2. cd to /webapp
3. Run Ganache (local blockchain)
4. Import accounts from ganache to Metamask wallet in chrome using seed phrase
5. run `python server.py`
6. Connect to `http://127.0.0.1:5000/home`
7. Start reading the browsing through the Decentralised Newspaper!

## Source Files

**The source code can be found in the webapp folder.**

**webapp/server.py is the Flask server with solc compiler**

**The HTML templates with inline javascripts (web3js) are found in webapp/templates**

**The smart contract can be found in webapp/Poll.sol**

**The testing data can be found in webapp/apptest.py**