Lab 3 – Real-World Interactions With Events and Oracles

COMP6452 Software Architecture for Blockchain Applications
Dilum Bandara

2023 Term 2

1 Learning Outcomes

In this lab, you will learn how a smart contract can send/receive data to/from the real world. After completing the lab, you will be able to:

- develop a smart contract that emits events on-chain that off-chain components could listen to
- develop an oracle that sends off-chain events into a smart contract
- develop programs that deploy a smart contract and interact with it to transfer/receive Ether
- test the oracle, smart contract, and user interactions

2 Introduction

Many use cases beyond purely digital assets need to interact with external systems. Thus, the validation of transactions on a blockchain might depend on external systems' states or vice versa. For example, a payment may be conditional on the physical properties of a product/service or the completion of a real-world activity like goods shipment. Typically, such external data are provided or certified by third parties.

Events are a way to log actions in a blockchain, and off-chain applications can subscribe and listen to these events. Events provide a mechanism to communicate and search for on-chain state changes (e.g., notifications arising from smart contract execution, changes to an account balance, and block generation) to external applications. For example, once a token redeem event is received from the blockchain, the custodian can release the underlying asset to the token owner. As transactions execute, events are recorded in the *transaction log*, a separate storage on a blockchain different from the ledger and transactions. Event logs are accessible using libraries like Web3.js. However, as events are stored separately on the blockchain, they are not accessible from within smart contracts.

Oracles are third-party services that introduce external systems' states into the closed blockchain execution environment. An oracle queries and verifies external data sources and then relays that data into the blockchain as a transaction forming an authoritative data record. For example, a reputed/trusted weather data provider can be designated as an oracle to report severe weather events like hail to a smart contract that automates agriculture insurance payouts. Events and oracles provide bidirectional data flow between blockchains and the external world. See oracle[1] and reverse oracle[2] patterns for more details.

The lab has two parts. In Section 3 and 4, we develop an escrow contract to illustrate how contracts relay data between the blockchain and the real world. It is then tested using the Remix IDE. In Section 5, we develop three programs to pragmatically handle events, report external data as an oracle, and simulate user actions.

[1] https://research.csiro.au/blockchainpatterns/general-patterns/interacting-with-the-external-world/oracle/

[2] https://research.csiro.au/blockchainpatterns/general-patterns/interacting-with-the-external-world/reverse-oracle/

3 Illustrative Example

Consider a scenario where parties involved in a transaction need to ensure that both the agreed product/service is delivered and payment is made. One party should not default on the transaction at the other party's expense. One way to solve this problem is to use an escrow service. Before making the transaction, the buyer transfers the payment to a third party called the *escrow*. The escrow holds the deposited funds until payment conditions are satisfied. If the escrow determines that the payment conditions are satisfied (e.g., goods are delivered as specified), it transfers the payment to the seller. Otherwise, the payment is returned to the buyer. Escrow services fees are relatively expensive and are used only for high-value or high-risk transactions.

Smart contacts could completely or partially replace the escrow[3], reducing transaction costs. For example, before the transaction, the buyer or seller could deploy an escrow smart contract specifying the settlement procedure and conditions. Second, the buyer transfers payment as cryptocurrency or tokens to the escrow contract. The escrow contract holds the payment until payment conditions are satisfied. Third, if the asset to be transferred is a token on the blockchain, the escrow contract can call the respective smart contract to validate the on-chain asset transfer automatically. Otherwise, it would need an oracle to confirm the off-chain transfer. Finally, if the payment conditions are satisfied, the escrow contract transfers the payment to the seller. Otherwise, the payment is returned to the buyer after a timeout. This lab assumes an off-chain asset where an oracle confirms its delivery to demonstrate the use of events and an oracle.

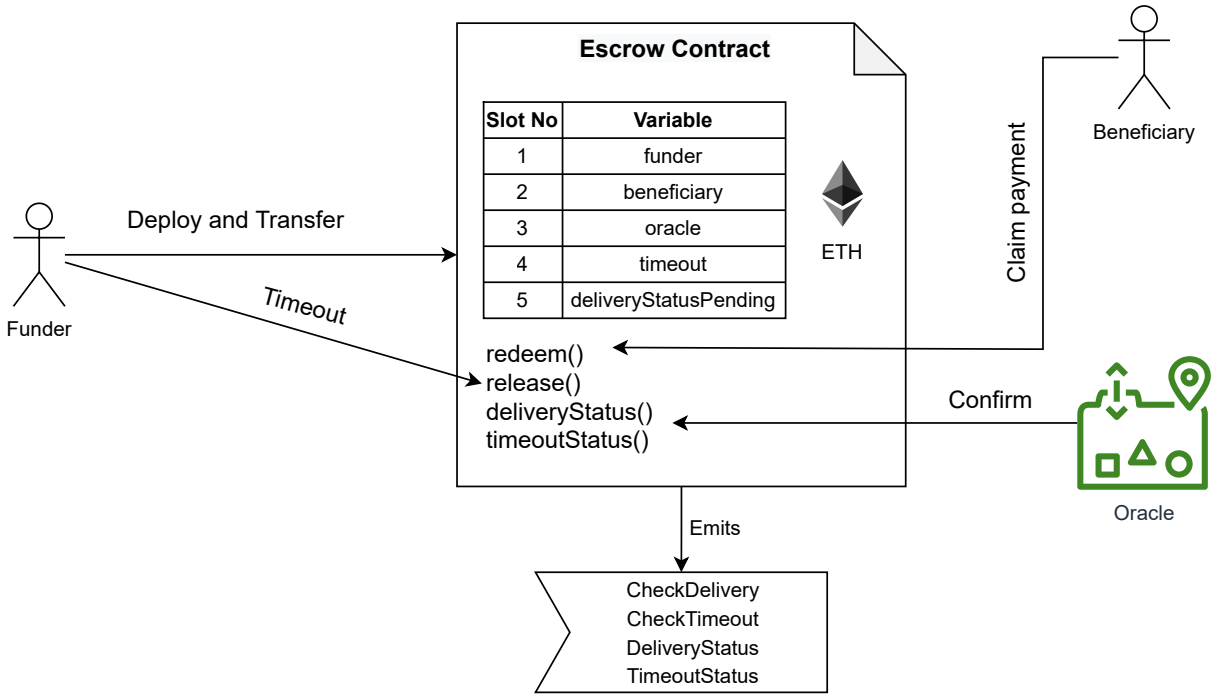

Figure 1: Escrow contract interactions.

Figure 1 illustrates the interactions between an escrow contract and its funder (i.e., buyer), beneficiary (i.e., seller), and oracle. The funder deploys the escrow contract in a single transaction and transfers the payment in Ether (ETH). The funder also specifies the beneficiary's address, designates an oracle by specifying its address, and sets a timeout. Note the state maintained in the escrow contract. Once the asset is delivered, the beneficiary calls the `redeem()` function to claim the payment. This function emits the `CheckDelivery` event that anyone can listen to. Once the event is received, the oracle checks whether the asset is delivered as expected. The oracle prompts its user to confirm delivery status to demonstrate human-in-the-loop decision-making. The oracle then reports the delivery status to the escrow contract by calling the `deliveryStatus()` function. If the delivery is confirmed, the escrow contract transfers the locked ETH to the beneficiary's address. The `DeliveryStatus` event is also emitted to inform any interested parties. As the escrow contract is no longer useful, the contract is destroyed. If the delivery

[3] https://research.csiro.au/blockchainpatterns/general-patterns/blockchain-payment-patterns/escrow-2/

status indicates an unsuccessful delivery, the `DeliveryStatus` event is emitted, but no funds will be transferred. The beneficiary may call redeem again until timeout.

Once the timeout is reached, and the asset is not yet delivered as expected, the funder can call the `release()` function. This will emit the `CheckTimeout` event. Once the event is received, the oracle checks whether the timeout is reached. We automate the timeout check as the time can be automatically checked using a web-based API (Application Programming Interface) from a reputed time source. The oracle reports the timeout status by calling the `timeoutStatus()` function. If the timeout is reached and payment is not released (i.e., the contract is still active), the escrow contract transfers locked ETH to the funder's address. The function also emits a `TimeoutStatus` event and destroys the contract.

4 Escrow Contract

The following code segment shows the Escrow contract. Lines 12-14 keep track of the funder, beneficiary, and oracle addresses. Line 15 keeps track of the timeout in AEST, represented as a Unix timestamp[4]. Unix timestamp represents time as an integer making it easier to process within a computer. Line 16 uses a Boolean variable to track whether the beneficiary requested to check the delivery status. Line 17 is another Boolean variable used to indicate whether the Escrow contract is in use. The Solidity `selfdestruct` function is now depreciated and will soon be removed from the EVM (Ethereum Virtual Machine). Therefore, we need to handle a contract's end-of-life on our own, and the use of a state variable like `inUse` is one of the recommended approaches.

```
1  /// SPDX-License-Identifier: UNLICENSED
2
3  pragma solidity ^0.8.0;
4
5  /// @title Escrow contract
6  /// This contract locks ETH until the delivery of asset
7  /// is confirmed or timeout is reached. To check the delivery
8  /// and wall clock time, we rely on an off-chain oracle.
9  /// @author Dilum Bandara, DFCRC
10
11 contract Escrow{
12     address public funder; // Owner address
13     address public beneficiary; // Beneficiary address
14     address public oracle; // Oracle address
15     uint256 public timeout; // Timeout in AEST as UNIX timestamp
16     bool private deliveryStatusPending; //is delivery proof pending
17     bool public inUse; // Contract in use
18
19     // Events informing contract activities
20     event CheckDelivery(address funder, address beneficiary);
21     event CheckTimeout(uint256 time);
22     event DeliveryStatus(bool status);
23     event TimeoutStatus(bool status);
24
25     /**
26      * @dev Constructor. Accept ETH as payment
27      *
28      * @param _beneficiary Beneficiary address
29      * @param _oracle Oracle address
30      * @param _timeout Timeout in AEST as UNIX timestamp
31      */
32     constructor (address _beneficiary, address _oracle, uint256 _timeout) payable{
33         funder = msg.sender; // Set escrow funder
34         beneficiary = _beneficiary;
35         oracle = _oracle;
36         timeout = _timeout;
37         deliveryStatusPending = false; // Proof not yet requested
38         inUse = true; // Contract in use
39     }
40
41     /**
42      * @dev Request to redeem as beneficiary. Once the request is made
43      * oracle is informed to report status via CheckDelivery event
44      */
45     function redeem() public{
```

[4] https://en.wikipedia.org/wiki/Unix_time

```
46          require(inUse); // Contract is still in use
47          require(msg.sender == beneficiary, 'Only beneficiary can call');
48
49          if(deliveryStatusPending) // Ignore if proof is pending
50              return;
51          deliveryStatusPending = true; // Mark as proof pending
52
53          emit CheckDelivery(funder, beneficiary); // Notify oracle
54      }
55
56      /**
57       * @dev Request to release funds as funder. Once the notification
58       * is made oracle is informed via CheckTime event
59       */
60      function release() public{
61          require(inUse); // Contract is still in use
62          require(msg.sender == funder, 'Only funder can call');
63
64          if (deliveryStatusPending) // Ignore if proof is pending
65              return;
66
67          emit CheckTimeout(timeout); // Notify oracle to check
68      }
69
70      /**
71       * @dev Oracle submit asset delivery status. If asset is delivered
72       * locked payment is released to the beneficiary and the
73       * contract is destroyed. Otherwise, do nothing
74       *
75       * @param isDelivered Is the asset delivered
76       */
77      function deliveryStatus(bool isDelivered) public{
78          require(inUse); // Contract is still in use
79          require(msg.sender == oracle, 'Only oracle can call');
80
81          emit DeliveryStatus(isDelivered);
82
83          if (isDelivered){ // Asset delivered
84              // Transfer all locked payment to beneficiary
85              payable(beneficiary).transfer(address(this).balance);
86              inUse = false; // Mark contract as no longer in use
87          }
88          deliveryStatusPending = false; // Delivery status check complete
89      }
90
91      /**
92       * @dev Oracle submit timeout status. If timeout reached
93       * locked payment is released to the funder and the
94       * contract is destroyed. Otherwise, do nothing
95       *
96       * @param isTimeout Is timeout reached
97       */
98      function timeoutStatus(bool isTimeout) public{
99          require(inUse); // Contract is still in use
100         require(msg.sender == oracle, 'Only oracle can call');
101
102         emit TimeoutStatus(isTimeout);
103
104         if (isTimeout){ // Escrow timeout
105             // Transfer all locked payment to funder
106             payable(funder).transfer(address(this).balance);
107             inUse = false; // Mark contract as no longer in use
108         }
109     }
110 }
```

Lines 20-23 define the four events the Escrow contract emits. The `CheckDelivery` event includes two arguments, namely the funder and beneficiary addresses. The `CheckTimeout` event includes the timeout. `DeliveryStatus` and `TimeoutStatus` contain a Boolean argument each to indicate the delivery and timeout status reported by the oracle.

Constructor (lines 32-39) accepts the beneficiary address, oracle addresses, and timeout while deploying the contract. Solidity requires any function that sends or receives ETH to be marked as `payable`, as

in line 32. Hence, the funder should send the payment in ETH while deploying the contract. The funder is assigned to the contract deploying party's address (line 33). The `DeliveryStatusPending` variable is initially set to `false` as the `redeem` function is not yet called. The `inUse` status is set to `true` as the contract is active as soon as it is deployed.

The beneficiary is expected to call the redeem function (lines 45-54) after delivering the asset off-chain. The `require` keyword in line 46 checks whether the contract is still active. Else, the function call will revert. The `require` keyword in line 47 ensures that only the beneficiary can execute the function. Even if another party calls the function, lines 85 and 106 ensure that locked ETH will be sent to either the funder or beneficiary depending on the delivery and timeout status. If the `deliveryStatusPending` is already set to `true` (i.e., the Escrow contract is waiting for oracle's delivery confirmation), the function returns without emitting the `CheckDelivery` event. Otherwise, the `deliveryStatusPending` is set to `true`, and the `CheckDelivery` event with the funder and beneficiary addresses is emitted.

Line 62 ensures only the funder can call the `release` function. If the Escrow contract is waiting for the oracle to confirm the delivery status (i.e., `deliveryStatusPending` is `true`), the function returns. While two transactions submitted by the same oracle will not get reordered in Ethereum (possible in some other blockchains), the oracle may receive or process events out of order. Therefore, `deliveryStatusPending` acts as a state variable that prevents the funder from releasing the payment while the delivery status is still pending. If the delivery status is not pending, the function emits the `CheckTimeout` event.

Only the oracle should call `deliveryStatus` and `timeoutStatus` functions to report delivery and timeout status, respectively. This is ensured by lines 79 and 100. Once the function is called, an event is emitted (line 81 or 102) to notify any party interested in the oracle's status report, e.g., the funder and beneficiary. If the delivery status is reported as `true`, in line 85, the payment is transferred to the beneficiary. `address(this).balance` retrieves the balance of this contract, i.e., Escrow contract. Then `payable(beneficiary).transfer` function is used to transfers that balance to the beneficiary. Finally, the contract is marked as no longer in use by setting `inUse` to `false`. If the delivery status is reported as `false`, the `deliveryStatusPending` is reset in line 88, allowing the `redeem` or `release` functions to be called.

Once the oracle calls the `timeoutStatus` function, an event is emitted to notify any party interested in the oracle's timeout status report. If the oracle confirms the timeout, in line 106, the payment is returned to the funder and the Escrow contract is marked as not in use. Otherwise, the rest of the function is not executed.

Step 1. Go to Remix IDE. Create a new contract and name it `Escrow.sol`. Then copy the above code. Compile the Escrow contract.

Step 2. Go to the `Deploy & run transactions` pane. Select `Remix VM (Shanghai)` as the `ENVIRONMENT`. Set the following constructor parameters and payment value:

- `_beneficiary` – Second pre-funded address from the `ACCOUNT` dropdown list
- `_oracle` – Third pre-funded address from the `ACCOUNT` dropdown list
- `_timeout` – Go to https://www.epochconverter.com and set a time at least three minutes into the future. Then click on the `Human date to Timestamp` button. Copy the Unix timestamp given as the `Epoch timestamp`. Paste the value into the constructor parameter on Remix IDE. Alternatively, you may use another website, `date '+%s'` command on a Unix/Linux-based system, or your favourite programming language to find the Unix time.
- `VALUE` – 5 or more ETH (make sure to set units as `Ether`)

Select the first pre-funded address from the `ACCOUNT` dropdown list. Then deploy the contract.

Step 3. Invoke respective getter functions and ensure the funder, beneficiary, oracle, and timeout values are properly set.

Step 4. Next, we request to redeem the payment as the beneficiary. Select the second pre-funded address from the `ACCOUNT` dropdown list to issue transactions as the beneficiary. Click on the `redeem` button.

Expand the transaction details on the Remix console and observe the details under the `logs` key. Note that the transaction emitted the `CheckDeliver` event with the funder and beneficiary addresses as arguments (listed as `args` and indexed as first and second arguments).

Step 5. Next, we submit the delivery status as the oracle. Select the third pre-funded address from the `ACCOUNT` dropdown list to issue transactions as the oracle. Let us assume the asset was not delivered as expected. Hence, enter `false` in the textbox next to the `deliveryStatus` button, and click the button.

Expand the transaction details on the Remix console and observe that the `status` argument of the `DeliveryStatus` event is set to `false`. Also, note that the beneficiary's account balance remains unchanged.

Step 6. Repeat Step 4 to redeem again as the beneficiary.

Step 7. Select the oracle's address from the `ACCOUNT` dropdown list. This time call the `deliveryStatus` function with `true` as input.

Note that the beneficiary's account balance has increased as it received the payment. Also, note that the Escrow contract's balance is now zero and `inUse` is set to `false`. Because the contract is now marked as not in use, if you try to invoke any functions other than getters, respective transactions should fail.

Step 8. Now that the contract is marked as not in use, we cannot check the timeout behaviour. For this, we need to deploy the Escrow contract again. Repeat Step 2 and 3.

Step 9. Select the second address in the `ACCOUNT` dropdown list, and request to redeem as the beneficiary.

Step 10. While the delivery status is pending, change back to the first address in the `ACCOUNT` dropdown list. Then request to release the payment as the funder (i.e., click on the `release` button). Note that the `CheckTimeout` event did not emit because the delivery status is pending.

Step 11. Change `ACCOUNT` to the oracle's address and submit the delivery status as `false`.

Step 12. change back to the first address in the `ACCOUNT` dropdown list. Then request again to release the payment as the funder.

Step 13. Change `ACCOUNT` to the oracle's address. To confirm the timeout, enter `true` in the textbox next to the `timeoutStatus` button, and click the button.

This confirms that the timeout has been reached. Consequently, the Escrow then releases the payments back to the funder, which we can validate by checking the funder's account balance. Also, note that the contract's ETH balance is zero.

5 Programmatically Handling Events and User Roles

5.1 Project Setup

For this section, we will use the sample code given in the `Lab3_Code.zip` file. As covered in Lab 2, ensure that NodeJS and Truffle Ganache (CLI is preferred) are installed. Start Ganache and start an Ethereum instance using the default configuration.

Step 14. Choose a suitable folder to create your custom project. Initialise the repository by executing the following commands using the terminal (part of the command starting with # is a comment). If you prefer `npm` over `yarn` to install NodeJS packages, use the relevant npm commands (see Lab 2).

```
cd Lab3
yarn init -y

#Use one of the following depending whether you have wget or curl to create the gitignore file
wget https://raw.githubusercontent.com/github/gitignore/master/Node.gitignore -O .gitignore
curl https://raw.githubusercontent.com/github/gitignore/master/Node.gitignore -o .gitignore

# Add typescript support, install dependencies, and configure typescript
yarn add -D typescript ts-node
yarn add -D @types/node
yarn add web3 solc fs-extra path axios # Install dependencies
```

```
npx tsc --init --rootDir src --outDir build --esModuleInterop --resolveJsonModule --lib es6 --module
    commonjs --allowJs true --noImplicitAny true

# Create source files for funder, beneficiary, oracle, and contract
mkdir src
mkdir src/funder && touch src/funder/index.ts
mkdir src/beneficiary && touch src/beneficiary/index.ts
mkdir src/oracle && touch src/oracle/index.ts
mkdir contracts
mkdir eth_providers && touch eth_providers/providers.json # Create provider entry
mkdir eth_accounts && touch eth_accounts/accounts.json # Create account entry
```

Step 15. Populate the following source files based on the Lab 3 sample files. Make sure to change the private keys on `accounts.json` to match the accounts in your Ganache blockchain node.

- `contracts/Escrow.sol`
- `eth_accounts/accounts.json`
- `eth_providers/providers.json`
- `src/solc-lib.ts`
- `src/util.ts`
- `src/beneficiary/index.ts`
- `src/funder/index.ts`
- `src/oracle/index.ts`

`solc-lib.ts` is similar to the one from Lab 2. `util.ts` has the utility class from Lab 2 that enables us to bump up the gas limit of transactions. It is used by `index.ts` of all three parties. We assume the beneficiary and oracle do not have access to the Escrow contract's ABI (Application Binary Interface); hence, they know only the function signatures/definitions of the functions they are expected to call. Same with events. Conversely, the funder has access to the ABI because it deploys the contract. Therefore, it uses the usual `call` and `send` functions as in Lab 2.

5.2 Funder

The funder's code is in `src/funder/index.ts`. Similar to Lab 2, we need to initialise the WebSocket provider and then use an account to emulate funder. These are achieved using the `initProvider` (lines 18-31) and `getAccount` (lines 38-49) functions.

The `release` function (lines 57-74) calls the Escrow contract's `release` function. It is called by the `setTimeout` built-in function on line 170, after a given number of milliseconds. We set the timeout slightly higher than the Escrow contract's timeout to prevent prematurely requesting to redeem due to small clock errors on your computer. This value can be set by adjusting the `timeoutDelta` variable on line 87.

The program assumes that the Escrow contract's constructor parameters and payment amount are passed via the command line. These values are accessed in lines 95-98. Ethereum expects the transaction value to be specified in Wei (10^{18} Wei = 1 ETH). Hence, line 97 converts the given ETH value to Wei.

In lines 101 to 161, we initialise the Web3 provider, create an account object using the private key (the funder uses the first account), compile the Escrow contract, save it into a file for future use, and deploy the contract.

In lines 163-170, after the timeout, we send a transaction to request the payment release. For this, we run a local timer that automatically calls the `release` function. Between lines 173-210, we listen/-subscribe to three events from the Escrow contract. We do not need to subscribe to the `CheckTimeout` event, as it is emitted in response to the `release` function called by the funder. Note that to call `contract.events.<event name>` function you need access to the contact object. This is set up using the ABI and contract address in line 163.

We get a subscription identifier when the connection is established to the Ethereum client. This can be used later to reconnect to an existing subscription. When a new event is received, the respective `data`

event is triggered. From the returned `event` value, we can extract event arguments such as the funder, beneficiary, and status reported by the oracle.

If the timeout status is *true*, in lines 202-204, we retrieve the funder's account balance using the `web3.eth.getBalance` function, which retrieves the balance of a given address.

5.3 Beneficiary

Compared to the funder's program, the beneficiary's `index.ts` does not deploy a contact. We assume it only knows the deployed Escrow contract's address, and the definition of the `redeem` function and events. The Escrow contract's address is passed to the program as a command line argument (line 61). The beneficiary uses the second Ganache account.

After initialising the Web3 provider, the program checks the Escrow contract's balance and beneficiary's address (lines 84-105). The former uses the `web3.eth.getBalance` function. The latter uses the `web3.eth.call` function. As the beneficiary does not have access to the Escrow's ABI, we only define the `beneficiary` getter function's signature (lines 89-93).

In lines 108-165, we subscribe to all events other than the `CheckDelivery` event, as it is emitted in response to the `redeem` function call. As the beneficiary does not have access to the ABI, the `web3.eth.subscribe` function is used to subscribe to events. EVM encodes an event using the hash of the event signature. Therefore, to subscribe to a specific event, we need to specify the event signature's hash. Otherwise, we will receive all events emitted by the blockchain node, which could be too much to handle on a production blockchain. Therefore, in line 110, we set the subscribe topic to the SHA3 hash of the `CheckTimeout` event signature. Alternatively, to decode an event's arguments, we must specify how they are structured. For example, in lines 112-115, we specify the `time` argument of the `CheckTimeout` event and its data type `uint256`.

Like the funder's code, once subscribed to an event, we get a subscription identifier. The `data` event is then triggered every time the subscribed event is received. We use the `web3.eth.abi.decodeLog` function to decode the event arguments. If the `DeliveryStatus` event indicates successful delivery of the asset, we also query the account balance of the beneficiary address on lines 142-144.

After subscribing to the three events, in lines 168-194, the beneficiary sends a transaction to invoke Escrow's `redeem` function. Because the ABI is unavailable, the `redeem` function's signature (see lines 168-173) is passed to the `sendTransaction` function to ensure proper encoding of inputs before issuing the transaction.

5.4 Oracle

The oracle code in `src/oracle/index.ts` is similar to the beneficiary code, except for submitting the delivery and timeout status. It also does not use Escrow's ABI but instead relies on the deployed contract's address and function signatures. The oracle uses the third account in Ganache.

In lines 91-160, the oracle subscribes to the `CheckDelivery` event. Once the event is received, it prompts the user to confirm the asset delivery status. For command line user interaction, we use the NodeJS `readline` module (see lines 108-156). Once prompted, a user can confirm the delivery status as `yes` or `no`. Consequently, the program calls Escrow's `deliveryStatus` function with delivery status marked as `true` or `false`. Such an oracle that relies on human inputs is typically called a "human oracle".

Alternatively, in lines 172-222, we automate the timeout check. Once the `CheckTimeout` event is received, the program makes an API call to a web-based time service at http://worldtimeapi.org. We specify Australia and Sydney as the country and city, respectively. You may need to change these to match your time zone. NodeJS `axios` module calls the API with the URL specifying the country and city. It returns a JSON object consisting of multiple time attributes, from which we extract the `unixtime`. Line 182 compares the Unix time against the timeout from the `CheckTimeout` event, and then invokes Escrow's `timeoutStatus` function with `true` or `false` as input.

5.5 Testing

Step 16. (optional) If you want to install the sample files without going through the above project setup and coding steps, run the following commands after extracting the `Lab3_code.zip` file.

```
yarn
```

Step 17. Use the following command to compile the TypeScript files, which generates JavaScript files in the `build` folder:

```
npx tsc
```

Step 18. Execute the funder's program that deploys the smart contract to the Ganache blockchain node using the following command (an example is given below):

```
node build/funder/index.js <beneficiary address> <oracle address> <payment in ETH> <timeout as Unix
    timestamp> # Command format
node build/funder/index.js 0x4c8e4b399B9B6214dcc943cB198EDA4dB43206bB 0
    x9C60b54046a52321B158011BEBEFF80478B56bee 10 1661682991 # Example
```

Make sure beneficiary and oracle addresses match the accounts on your Ganache instance. Also, set payment amount and timeout as command line arguments. You should see output like this:

```
---- Funder ----
Connected to Web3 provider.
Funder running as account acc0 with address 0x3ffe2E90eFc2632AB03E80F67A715D554ef31AC2
Writing: Escrow.json to /Users/Documents/Workspace/tmp/Lab3_Code/build/funder
Contract compiled
Contract deployed at address: 0xD978010f1E61a17F2e8eA3A470399381a677e366
Escrow timeout will be called in 63 seconds
```

Note down the Escrow contract's address.

Step 19. Open a new terminal instance. Launch the oracle's program using the following command (an example is given below):

```
node build/oracle/index.js <Escrow contract address> # Command format
node build/oracle/index.js 0xD978010f1E61a17F2e8eA3A470399381a677e366 # Example
```

Note that we need to pass the Escrow contract's address as a command line argument. You should see output like this:

```
---- Oracle ----
Connected to Web3 provider.
Oracle running as account acc2 with address 0xF1e3b61d647d5F0eC2897691Fa0fF97ea73327a8
```

Step 20. Open another terminal instance. Launch the beneficiary's program using the following command (an example is given below):

```
node build/beneficiary/index.js <Escrow contract address> # Command format
node build/beneficiary/index.js 0xD978010f1E61a17F2e8eA3A470399381a677e366 # Example
```

Note that we need to pass the Escrow contract's address as a command line argument. You should see output like the following:

```
---- Beneficiary ----
Connected to Web3 provider.
Beneficiary running as account acc1 with address 0x10dC5729e31aD7083950994A2Bd17a7Fb0bbfB30
Escrow contract at address 0xD978010f1E61a17F2e8eA3A470399381a677e366 has a balance of 2 ETH
Beneficiary address on escrow: 0x00000000000000000000000010dc5729e31ad7083950994a2bd17a7fb0bbfb30
Requested to redeem funds.
```

Verify the Escrow contract's account balance and beneficiary address.

Step 21. Now that the beneficiary has requested to `redeem` the payment, the funder and oracle should have received the `CheckDelivery` event. Return to the funder's terminal and check for a message like the following:

```
Event CheckDelivery received with funder 0x3ffe2E90eFc2632AB03E80F67A715D554ef31AC2 and beneficiary: 0
    x10dC5729e31aD7083950994A2Bd17a7Fb0bbfB30
```

Step 22. Return to the oracle's terminal. It should prompt you to confirm the delivery status as follows:

```
Event CheckDelivery received with funder 0x3ffe2E90eFc2632AB03E80F67A715D554ef31AC2 and beneficiary: 0
    x10dC5729e31aD7083950994A2Bd17a7Fb0bbfB30
Is the asset delivered?
```

Enter `yes`. This should emit the `DeliveryStatus` event. Return to the beneficiary terminal, where you should see a message like the following:

```
Event DeliveryStatus received with status true
Beneficiary has a balance of 103.9998032359999 ETH
```

Step 23. Stop all three programs by pressing `CTRL+C`. Repeat Steps 17-22, while allowing the contract timeout to occur such that payment is returned to the funder.

Step 24. You will notice that either the funder or beneficiary program will crash depending on who redeems or releases the last. This is because the last transaction will fail as the Escrow contract is disabled after transferring the funds. Update your code to handle this error.