Designing Tellor as a Push Oracle

Christopher Pondoc

cpondoc@tellor.io

Abstract

In this litepaper, I describe an implementation for utilizing Tellor as a push oracle. The implementation includes developing two interfaces: one for the push oracle itself, and one for the user of the oracle contract. Outside of the oracle providing data to the user contract, mechanisms are put in place to provide incentives for the oracle, including payment in Tellor Tributes (TRB) to the oracle for the data as well as compensation for the gas used in retrieving the off-chain data. Finally, I compare pull and push oracle designs, as well as specific security vulnerabilities for each.

1 Introduction

The construct of a "push" oracle was first introduced by Alan Lu back in June of 2018, as a part of EIP-1154 [1], which strove to create a standard interface for oracles. Within the proposal, Alan Lu outlined the structure of a push oracle: an oracle that pushes the data directly into smart contracts that utilize the oracle. In other words, the user – or in his case, "consumer" – contract doesn't 'call' a function to grab data; the interface instead provides functionality to receive the data from an approved oracle. Below was Alan's proposed interface for a consumer of an oracle:

```
interface OracleConsumer {
    function receiveResult(bytes32 id, bytes result) external;
}
```

At a programmatic level, the function must also include several require statements:

- Ensure that msg.sender was an authorized oracle to provide data to the smart contract
- Ensure that the same ID had not been called twice
- Ensure that the contract could handle the data to begin with

2 Pull Oracles

The push mechanism is different from a pull oracle, which allows users of the oracle to call a function directly. When looking at the proposed interface, two differences are apparent. First, the pertinent function resides within the oracle smart contract as opposed to the consumer contract. Furthermore, the function returns the value of the result rather than including the value as a function parameter:

2.1 Constructor

First and foremost, the contract inherits the 'UsingTellor' smart contract. The contract's constructor consists of an address of an eligible Tellor oracle contract – for testing purposes, this would often be an instance of Tellor playground – a mapping of data IDs, and a corresponding mapping of smart contract addresses for each liquidity pool. The constructor requires that the amount of data IDs and contracts are equal, and then assigns the internal mapping of price feeds to liquidity pools using a loop.

```
interface Oracle {
    function resultFor(bytes32 id) external view returns (bytes result);
}
```

Here, Alan proposed that the function would include two require statements:

- Ensure that the data for the specific ID is available
- Ensure that the result is the same for an ID after the result is available

3 Tellor Implementation

Part of the benefit for creating a standard interface would be the ease of interchanging between the two oracle designs: a push oracle could easily be made into a pull oracle, and vice versa. As a case study, I decided to look at the Tellor oracle and turn it from one type into the other.

Tellor's decentralized oracle can be utilized by installing the 'usingtellor' Node.js package. To use the Tellor smart contracts, a sample Solidity script is provided, which allows for easy integration:

```
import "usingtellor/contracts/UsingTellor.sol";
import "usingtellor/contracts/TellorPlayground.sol";

contract BtcPriceContract is UsingTellor {

  //This Contract now have access to all functions on UsingTellor

  uint256 btcPrice;
  uint256 btcRequestId = 2;

  constructor(address payable _tellorAddress) UsingTellor(_tellorAddress) public
  ↪  {}

  function setBtcPrice() public {
    bool _didGet;
    uint _timestamp;
    uint _value;

    (_didGet, btcPrice, _timestamp) = getCurrentValue(btcRequestId);
  }
}
```

3.1 Push Oracle Contract

First, let's take a look at the push oracle contract. The push oracle contract is initialized using a constructor that takes both the address of a Tellor contract – in testing conditions, this is typically an instance of Tellor Playground – and the address of an ERC-20 token (this will be the address of Tellor Tributes (TRB) in the future, but more on that later). The contract itself inherits the 'UsingTellor' contract, so a call to its internal constructor is also made.

```
contract TellorPushOracle is UsingTellor {
    using SafeMath for uint256;
    // Storage
    ERC20 token; // Token to transfer as payment to the oracle

    // Functions
    /**
     * @dev Sets up UsingTellor oracle by initializing with address and defines
     address of ERC20 token used
     * @param _tellorAddress address of Tellor oracle or Tellor Playground
↪   instance
     */
    constructor(address payable _tellorAddress, address _tokenAddress)
    ↪   UsingTellor(_tellorAddress) {
        token = ERC20(_tokenAddress);
}
```

The primary function for the push oracle contract is ‘pushNewData,’ which performs the push operation to put new data into the oracle user contract. The function has two parameters – a tellorID for the data ID, and a payable contract for the user – and serves two primary purposes, First and foremost, ‘pushNewData’ retrieves data for the user. To do this, the function calls the ‘getCurrentValue’ function of the UsingTellor smart contract, allowing it to get data from the Tellor oracle. If the data was not received, the function throws an error. If the data was received, the function then creates a new instance of a Tellor user contract, and calls the ‘receiveResult’ function on the user to push new data into the contract.

```
// Grab current value from Tellor oracle and ensure data was retrieved
(bool ifRetrieve, uint256 value, ) = getCurrentValue(_tellorID);
require(ifRetrieve == true, "Data was not retrieved!");
```

3.2 Calculation of Criteria and Thresholds

The second purpose of the function is to calculate the amount of gas that should be paid back to the oracle. Normally, with a pull oracle, the user contract pays its own gas for an external function call to the oracle contract. However, because the push mechanism requires that the oracle contract directly push the values into the user contract, the contract should calculate the amount of gas that is utilized in making the call to get the value, and passing that value into the user contract for means of repayment. This is done by determining the initial amount of gas at the beginning of the contract, and then subtracting what is left after calling ‘getCurrentValue’. This value is then passed in as a parameter to ‘receiveResult’.

```
function pushNewData(uint256 _tellorID, address payable _userContract) external
↪   payable {
    uint256 initialGas = gasleft(); // Get initial gas

    // Grab current value from Tellor oracle and ensure data was retrieved
    (bool ifRetrieve, uint256 value, ) = getCurrentValue(_tellorID);
    require(ifRetrieve == true, "Data was not retrieved!");

    uint256 gasDifference = initialGas.sub(gasleft()); // Calculate difference
    ↪   in gas

    // Pushes data to Tellor user contract using receiveResult function
    TellorPushUser tellorUser = TellorPushUser(_userContract);
    tellorUser.receiveResult(_tellorID, value, gasDifference);
}
```

Finally, the oracle contract contains several functions to get the ether and tribute balance of the contract, as well as functions that handle payment of ether ('receive' and 'fallback').

```
/**
 * @dev Returns amount of ether the smart contract holds
 */
function getEtherBalance() public view returns (uint) {
    return address(this).balance;
}

/**
 * @dev Function to receive Ether. msg.data must be empty
 */
receive() external payable {}

/**
 * @dev Fallback function is called when msg.data is not empty
 */
fallback() external payable {}
```

3.3 User Contract

Now, let's take a look at the implementation of the user of an oracle contract. The contract inherits the interface 'ITellorPushUser', which defines a set of functions that should be common to all users of push oracles (as per EIP-1154 standards), and also defines a couple of local variables. These include:

- The address of an oracle that is approved to provide data to the smart contract (can be changed to a mapping of different oracle contracts, if the diversification of sources is emphasized)
- The last request/data ID updated
- The internal oracle values, which are a mapping of request IDs (uint256s) to values (uint256s)
- An ERC20 token that determines how to repay the oracle for using the oracle

The contract itself is initialized using a constructor that takes the address of an approved oracle, as well as the address of an ERC-20 token. These will be the Tellor Push oracle and Tellor Tribute, respectively.

```
contract TellorPushUser is ITellorPushUser {
    // Storage
    address approvedOracle; // address of oracle approved to provide data
    uint256 lastRequestId; // last request ID updated by the oracle
    mapping(uint256 => uint256) internalOracle; // mapping of values to grab
     ↪  from
    ERC20 token; // Token to transfer as payment to the oracle

    // Functions
    /**
     * @dev Constructor solely defines the contract address approved to be an
↪   oracle
     * @param _oracleAddress address of approved oracle smart contract
     */
    constructor(address payable _oracleAddress, address _tokenAddress) {
        approvedOracle = _oracleAddress;
        token = ERC20(_tokenAddress);
    }
}
```

The primary function that is implemented is the ‘receiveResult’ function, which takes a requestID, an oracle value, and the gas to repay as parameters. First and foremost, the function calculates the total gas to repay by multiplying the difference in gas by the transaction gas price. If the contract does not have enough ether to repay the contract, an error is thrown.

```
function receiveResult(uint256 _requestID, uint256 _oracleValue, uint256
↪   _gasDifference) override external {
    // Calculate gas to refund and make sure that contract has enough ether
    uint256 gasRefund = _gasDifference * tx.gasprice;
    require(address(this).balance > gasRefund, "The contract does not have
     ↪  enough ether to pay back the oracle!");
```

Per the EIP-1154 standards, two other ‘require’ statements are implemented: one ensures that the sender is an approved oracle address, and the other ensures that the current request ID being updated is not the same as the last updated.

```
// Require statements per EIP-1154: Oracle Interface
require(msg.sender == approvedOracle, "The address is not an approved oracle!");
require(lastRequestId != _requestID, "This request ID has been called before!");
```

Finally, the function updates the last request ID called, updates the value of the internal oracle, and ensures that the contract has enough ether and tributes to repay the contract.

```
// Update last request ID and mapping of values
lastRequestId = _requestID;
internalOracle[lastRequestId] = _oracleValue;

// Check if funds are sufficient, then transfer tokens over
require(token.balanceOf(address(this)) > 100, "The User of Tellor does not have
↪   enough tributes!");
token.transfer(msg.sender, 100);

// Check if funds are sufficient, then pay for gas
(bool sent,) = approvedOracle.call{value: gasRefund}("");
require(sent, "The user contract failed to send Ether!");
```

The rest of the contract consists of getters to retrieve oracle values and balances from the smart contract, as well as functions to handle the payment of ether (‘receive’ and ‘fallback’, once more).

```
/**
 * @dev Returns value of specific request ID stored by a user of Tellor
 * @param _requestID the ID of the data to be returned
 * @return uint256 the value of the oracle data
 */
function getUserValue(uint256 _requestID) external view returns (uint256) {
    return internalOracle[_requestID];
}

/**
```

```
 * @dev Returns Tribute balance of the Tellor User contract
 * @return uint256 the number of ERC 20 tokens the specific user has
 */
function getTributeBalance() external view returns(uint256) {
    return token.balanceOf(address(this));
}

/**
 * @dev Returns amount of ether the smart contract holds
 */
function getEtherBalance() public view returns (uint) {
    return address(this).balance;
}

/**
 * @dev Function to receive Ether. msg.data must be empty
 */
receive() external payable {}

/**
 * @dev Fallback function is called when msg.data is not empty
 */
fallback() external payable {}
```

4 Conclusion

Overall, the Tellor oracle transitioned nicely from a pull into a push oracle. Note that the mechanisms for repaying of gas, as well as the numerous constraints on handling data provided from the push oracle make it much more cumbersome to implement. However, for contracts looking for uncalled updates, the mechanism makes sense.

4.1 Acknowledgments

Thank you to Nick Fett, Tally Wiesenberg, and Brenda Loya for providing feedback and helping me brainstorm the Oracle design. And of course, thank you to the entire Tellor team for the opportunity to work with them during the Summer of 2021.

5 Sources

Alan Lu, "EIP-1154: Oracle Interface," Ethereum Improvement Proposals, no. 1154, June 2018. [Online serial].
Available: https://eips.ethereum.org/EIPS/eip-1154.

6 Contact

- For more information on my work, check out at github.com/cpondoc.
- For more information on Tellor, check out tellor.io.
- This project idea came from the Tellor Bounties page. To check out other bounties, go to tellor.io/bounties.