Town Crier

Release 1.0

Fan Zhang

Jul 29, 2021

Contents:

CHAPTER 1

How Town Crier Works

1.1 The Big Picture

Town Crier (TC) connects the authenticated data from HTTPS websites to smart contracts. TC works in a request-response fashion—the client (your fantastic smart contract that needs data) submit *queries* to TC, from which responses are furnished.

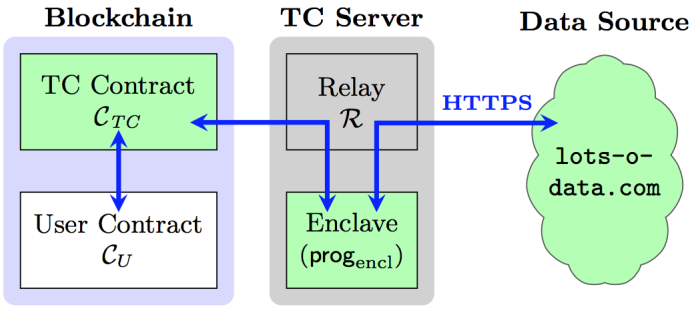

Fig. 1: TC serves as a bridge between authenticated data on the web and the blockchain.

1.2 A Little Bit More Details

Behind the scenes, TC has two components that work together to serve queries from client contracts:

TC Contract *The frontend.* A smart contract deployed on the blockchain that is responsible to interface with client contracts.

TC Server *The backend.* A SGX-protected process that actually handles the queries picked up by the fronend. When the TC contract receives a query from a client contract, the TC server fetches the requested data from the website and relays it back to the requesting contract.

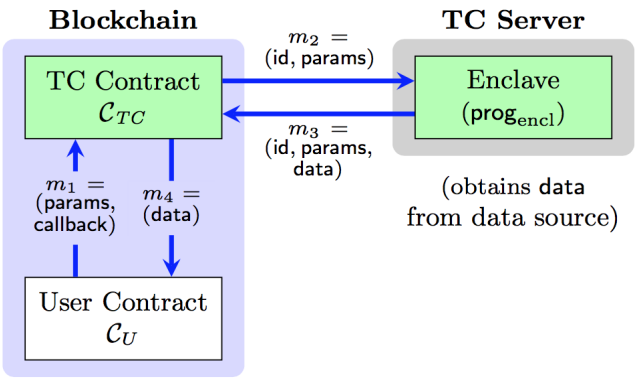

Fig. 2: TC Contract and TC Server.

1.3 Security of TC

Query processing happens inside an SGX-protected environment known as an **enclave**. The requested data is fetched via a TLS connection to the target website that terminates inside the enclave. SGX protections prevent even the operating system of the server from peeking into the enclave or modifying its behavior, while use of TLS prevents tampering or eavesdropping on communications on the network.

Town Crier can optionally ingest an *encrypted* query, allowing it to handle *secret query data*. For example, a query could include a password used to log into a server or secret trading data. TC's operation in an SGX enclave ensures that the password or trading data is concealed from the TC operator (and everyone else).

CHAPTER 2

Get Started: Write Your First TC-powered Contract

The TC contract has a very simple API for your contract to call. Below we present the generic API, and then we'll go through an example (*Play with TC on Rinkeby*) with detailed explanation.

2.1 Submit queries via `request`

An application contract sends queries to TC by calling the `request` function.

```
request(uint8 requestType, address callbackAddr,
      bytes4 callbackFID, uint256 timestamp,
      bytes32[] requestData) public payable returns(int256);
```

Parameters:

- `requestType`: indicates the query type (see below).
- `callbackAddr`: the address of the recipient contract (see below).
- `callbackFID`: the callback function selector (see below).
- `timestamp`: reserved. Unused for now.
- `requestData`: data specifying query parameters. The format depends on the query type.

Warning: Calling contracts must prepay the gas cost incurred by the Town Crier server in delivering a response to the application contract properly. The calling contract is responsible to set `msg.value` properly.

Return: The function returns an `int256` value denoted as `requestId`.

- If `requestId > 0`, then the request is successfully submitted. The calling contract can use `requestId` to check the response or status.
- If `requestId = -2^250`, then the request fails because the requester didn't send enough fee to the TC Contract.

- If `requestId = 0`, then the TC service is suspended due to some internal reason. No more requests or cancellations can be made but previous requests will still be responded to by TC.
- If `requestId < 0 && requestId != -2^250`, then the TC Contract is upgraded and requests should be sent to the new address `-requestId` (negative of `requestId`).

2.2 Canceling requests via `cancel`

```
cancel(uint64 requestId) public returns(bool);
```

Unprocessed requests can be canceled by calling function `cancel(requestId)`. The fee paid by the requester is refunded (minus processing costs, denoted as cancellation fee).

For more details about how Town Crier contract works, you can look at the source code of the contract TownCrier.sol.

2.3 Receiving responses from TC

To receive a response from TC, the requester need to specify the recipient contract as well as the recipient function.

Warning: Very importantly, TC requires that the recipient function to have the following signature.

```
function FUNCTION_NAME(uint64 requestId, uint64 error, bytes32 respData) public;
```

This is the function that will be called by the TC Contract to deliver the response from TC server.

To do so, you should set the `callbackFID` parameter to `bytes4(sha3("FUNCTION_NAME(uint64,uint64,bytes32)"))`, namely the selector of your `FUNCTION_NAME` function.

Parameters: `error` and `respData`.

- If `error = 0`, the request has been successfully processed and the calling contract can then safely use `respData`. The fee paid is consumed by TC.
- If `error = 1`, the provided request is invalid or cannot be found on the website. In this case, similarly, the fee is consumed by TC.
- If `error > 1`, then an error has occured in the Town Crier server. In this case, the fee is fully refunded but the transaction fee (for making this call).

CHAPTER 3

Play with TC on Rinkeby

Note: A version of Town Crier smart contract has been deployed on the Rinkeby testnet. To show how to use Town Crier, we present a skeleton `Application` Contract that does nothing other than sending queries, logging responses and cancelling queries. The Application contract has also been deployed on Rinkeby (https://rinkeby.etherscan.io/address/0x20e63d9683a75ef73e6174298354f8b016878de3).

The source code of the application contract can be found here. Now we go through the contract code line by line.

3.1 Preliminaries

First, you need to annotate your contract with the version pragma:

```
pragma solidity ^0.4.9;
```

Second, you need to include in your contract the function declaration of the `TownCrier` Contract so that the application contract can call those functions with the address of the `TownCrier` Contract.

```
contract TownCrier {
    function request(uint8 requestType,
             address callbackAddr,
             bytes4 callbackFID,
             uint timestamp,
             bytes32[] requestData) public payable returns (uint64);
    function cancel(uint64 requestId) public returns (int);
}
```

Note: You do not need to include `response()` here because an appilcation contract should not make a function call to it but wait for being called by it.

3.2 Outline of `Application.sol`

Let's look at the layout of the `Application` Contract:

```
contract Application {
    event Request(int64 requestId, address requester, uint dataLength, bytes32[]␣
↪data);
    event Response(int64 requestId, address requester, uint64 error, uint data);
    event Cancel(uint64 requestId, address requester, bool success);

    bytes4 constant TC_CALLBACK_FID = bytes4(sha3("response(uint64,uint64,bytes32)"));

    address[2**64] requesters;
    uint[2**64] fee;

    function() public payable;
    function Application(TownCrier tcCont) public;
    function request(uint8 requestType, bytes32[] requestData) public payable;
    function response(uint64 requestId, uint64 error, bytes32 respData) public;
    function cancel(uint64 requestId) public;
}
```

- The events `Request()`, `Response` and `Cancel()` keeps logs of the `requestId` assigned to a query, the response from TC and the result of a cancellation respectively for a user to fetch from the blockchain.
- The constant `TC_CALLBACK_FID` is the first 4 bytes of the hash of the function `response()` that the `TownCrier` Contract calls when relaying the response from TC. The name of the callback function can differ but the three parameters should be exactly the same as in this example.
- The address array `requesters` stores the addresses of the requesters.
- The uint array `fee` stores the amounts of wei requesters pay for their queries.

As you can see above, the `Application` Contract consists of a set of five basic functions:

3.2.1 Default Function

```
function() public payable;
```

This fallback function must be payable so that TC can provide a refund under certain conditions. The fallback function should not cost more than 2300 gas, otherwise it will run out of gas when TC refunds ether to it. In our contract, it simply does nothing.

```
function() public payable {}
```

3.2.2 Constructor

```
function Application(TownCrier tc) public;
```

This is the constructor which registers the address of the TC Contract and the owner of this contract during creation so that it can call the `request()` and `cancel()` functions in the TC contract.

```
TownCrier public TC_CONTRACT;
address owner;

function Application(TownCrier tcCont) public {
    TC_CONTRACT = tcCont;
    owner = msg.sender;
}
```

3.2.3 Submitting Requests

```
function request(uint8 requestType, bytes32[] requestData) public payable;
```

A user calls this function to send a request to the `Application` Contract. This function forwards the query to the `request()` of the TC Contract by

```
requestId = TC_CONTRACT.request.value(msg.value)(requestType, TC_CALLBACK_ADD, TC_
↪CALLBACK_FID, timestamp, requestData);
```

`msg.value` is the fee the user pays for this request. `TC_CALLBACK_ADD` is the address of the callback contract. or `this` for the current contract. `TC_CALLBACK_FID` is the first 4 bytes of the hash of the callback function signature, as defined above.

```
uint constant MIN_GAS = 30000 + 20000;
uint constant GAS_PRICE = 5 * 10 ** 10;
uint constant TC_FEE = MIN_GAS * GAS_PRICE;

function request(uint8 requestType, bytes32[] requestData) public payable {
    if (msg.value < TC_FEE) {
        // If the user doesn't pay enough fee for a request,
        // we should discard the request and return the ether.
        if (!msg.sender.send(msg.value)) throw;
        return;
    }

    int requestId = TC_CONTRACT.request.value(msg.value)(requestType, this, TC_
↪CALLBACK_FID, 0, requestData);
    if (requestId == 0) {
        // If the TC Contract returns 0 indicating the request fails
        // we should discard the request and return the ether.
        if (!msg.sender.send(msg.value)) throw;
        return;
    }

    // If the request succeeds,
    // we should record the requester and how much fee he pays.
    requesters[uint64(requestId)] = msg.sender;
    fee[uint64(requestId)] = msg.value;
    Request(int64(requestId), msg.sender, requestData.length, requestData);
}
```

Warning: Developers need to send enough fee.

TC requires at least **3e4** gas for all the operations (besides calling the callback function). The gas price is set to **5e10 wei**. So the caller should pay at least **(3e4 + callback_gas) * 5e10 wei**. Otherwise the `request` call will

fail (and the TC Contract will return 0 as `requestId`). Developers should handle this failure.

For our `Application.sol`, the callback function (`response`) costs about 2e4 gas, so the caller should pay no less than (3e4 + 2e4) * 5e10 = 2.5e15 wei (denoted as `TC_FEE`).

Note: TC server sets the gas limit as **3e6** when sending the response to the TC Contract. If a requester paid more gas than that, the excess ether will not be used for the callback function. It will go directly to the SGX wallet. This is a way to offer a tip for the Town Crier service.

3.2.4 Receiving Responses

```
function response(uint64 requestId, uint64 error, bytes32 respData) public;
```

This is the function to be called by the TC Contract to deliver the response from TC server. The selector for this function is passed to the request call. See *Submitting Requests*.

```
function response(uint64 requestId, uint64 error, bytes32 respData) public {
    // If the response is not sent from the TC Contract,
    // we should discard the response.
    if (msg.sender != address(TC_CONTRACT)) return;

    address requester = requesters[requestId];
    // Set the request state as responded.
    requesters[requestId] = 0;

    if (error < 2) {
        // If either TC responded with no error or the request is invalid by the␣
↪requester's fault,
        // public the response on the blockchain by event Response().
        Response(int64(requestId), requester, error, uint(respData));
    } else {
        // If error exists by TC's fault,
        // fully refund the requester.
        requester.send(fee[requestId]);
        Response(int64(requestId), msg.sender, error, 0);
    }
}
```

Warning: Since the gas limit for sending a response back to the TC Contract is set as **3e6** by the Town Crier server, as mentioned above, the callback function should not consume more gas than that. Otherwise the callback function will run out of gas and fail. The TC service does not take responsibility for such failures, and treats queries that fail in this way as successfully responded to.

To estimate how much gas the callback function costs, you can use web3.eth.estimateGas.

3.2.5 Cancellation

```
function cancel(uint64 requestId) public;
```

This function calls the `cancel()` function of the TC Contract, to cancel a unprocessed request.

```
uint constant CANCELLATION_FEE = 25000 * GAS_PRICE;

function cancel(uint64 requestId) public {
    // If the cancellation request is not sent by the requester himself,
    // discard the cancellation request.
    if (requestId == 0 || requesters[requestId] != msg.sender) return;

    bool tcCancel = TC_CONTRACT.cancel(requestId);
    if (tcCancel) {
        // If the cancellation succeeds,
        // set the request state as cancelled and partially refund the requester.
        requesters[requestId] = 0;
        if (!msg.sender.send(fee[requestId] - CANCELLATION_FEE)) throw;
        Cancel(requestId, msg.sender, true);
    }
}
```

TC charges **2.5e4 * 5e10 = 1.25e15 wei**, denoted as `CANCELLATION_FEE` here, for cancellation. In this function a user is partially refunded `fee - CANCELLATION_FEE`. A developer must carefully set a cancelled flag for the request before refunding the requester in order to prevent reentrancy attacks.

3.3 Send queries to Application.sol

You can play with the `Application.sol` deployed on Rinkeby testnet, at 0xdE34AfC49b8A15bEb76A6E942bD687143C1574B6.

Assuming we're at the geth console loaded with the following script. You can find a script for this purpose here.

```
function createApp(tc) {
    unlockAccounts();
    var tradeContract = App.new(
        tc, {
            from: tcDevWallet,
            data: "0x" + compiledContract.contracts["Application"].bin,
            gas: gasCnt
        },
        function (e, c) {
            if (!e) {
                if (c.address) {
                    console.log('Application created at: ' + c.address)
                }
            } else {
                console.log('Failed to create Application contract: ' + e)
            }
        });
    return tradeContract;
}

function request(contract, type, requestData) {
    unlockAccounts();
    contract.request.sendTransaction(type, requestData, {
        from: tcDevWallet,
        value: 3e15,
        gas: gasCnt
```

(continues on next page)

(continued from previous page)

```
    });
    return "Request sent!";
}

function watch_events(contract) {
    var his = contract.allEvents({fromBlock: 0, toBlock: 'latest'});
    var events;
    his.get(function (error, result) {
        if (!error) {
            console.log(result.length);
            for (var i = 0; i < result.length; ++i) {
                console.log(i + " : " + result[i].event);
            }
            events = result;
        } else {
            console.log("error");
            events = "error";
        }
    });
    return events;
}
```

Let's try to trigger Application.sol to query for bitcoin price (from coinmarketcap.com) and Bitcoin Fee.

First, create an instance of Application.sol.

```
> var App = web3.eth.contract(JSON.parse("[{"constant":false,"inputs":[{"name":
↪"requestType","type":"uint8"},{"name":"requestData","type":"bytes32[]"}],"name":
↪"request","outputs":[],"payable":true,"stateMutability":"payable","type":"function"}
↪,{"constant":false,"inputs":[{"name":"requestId","type":"uint64"}],"name":"cancel",
↪"outputs":[],"payable":false,"stateMutability":"nonpayable","type":"function"},{
↪"constant":true,"inputs":[],"name":"TC_CONTRACT","outputs":[{"name":"","type":
↪"address"}],"payable":false,"stateMutability":"view","type":"function"},{"constant
↪":false,"inputs":[{"name":"requestId","type":"uint64"},{"name":"error","type":
↪"uint64"},{"name":"respData","type":"bytes32"}],"name":"response","outputs":[],
↪"payable":false,"stateMutability":"nonpayable","type":"function"},{"inputs":[{"name
↪":"tcCont","type":"address"}],"payable":false,"stateMutability":"nonpayable","type":
↪"constructor"},{"payable":true,"stateMutability":"payable","type":"fallback"},{
↪"anonymous":false,"inputs":[{"indexed":false,"name":"requestId","type":"int64"},{
↪"indexed":false,"name":"requester","type":"address"},{"indexed":false,"name":
↪"dataLength","type":"uint256"},{"indexed":false,"name":"data","type":"bytes32[]"}],
↪"name":"Request","type":"event"},{"anonymous":false,"inputs":[{"indexed":false,"name
↪":"requestId","type":"int64"},{"indexed":false,"name":"requester","type":"address"},
↪{"indexed":false,"name":"error","type":"uint64"},{"indexed":false,"name":"data",
↪"type":"uint256"}],"name":"Response","type":"event"},{"anonymous":false,"inputs":[{
↪"indexed":false,"name":"requestId","type":"uint64"},{"indexed":false,"name":
↪"requester","type":"address"},{"indexed":false,"name":"success","type":"bool"}],
↪"name":"Cancel","type":"event"}]"));
> app = App.at("0xdE34AfC49b8A15bEb76A6E942bD687143C1574B6");
```

Now, send a few requests!

```
> request(app, 2, []); // get current bitcoin transaction fee
> request(app, 5, ['bitcoin']);"; // get current bitcoin price
```

To see the responses (and the requests), examine the log:

```
> watch_events(app);
```

You'll see something like this for bitcoin transaction fee query:

```
9 : Request
{
   "args":{
      "data":[

      ],
      "dataLength":"0",
      "requestId":"5",
      "requester":"0x8f108aab17e3b90f6855a73349511f5944b7e146"
   },
   "blockNumber":2182246,
   "transactionHash":
↪"0x027f8b992b65b58f1aa2191e6ae55d1c074cdaa475a71823d1c879ddc8cbae79",
}
10 : Response
{
   "args":{
      "data":"100", // fastestFee=100 from https://bitcoinfees.earn.com/api/v1/fees/
↪recommended
      "error":"0",
      "requestId":"5",
      "requester":"0x8f108aab17e3b90f6855a73349511f5944b7e146"
   },
   "blockNumber":2182248,
   "transactionHash":
↪"0x630d8f7fae392c2ff6a0115956e72cad5fae4e008f3ef5e543d02c112a7d5cf5",
}
```

For bitcoin price query:

```
11 : Request
{
   "args":{
      "data":[
         // ascii of 'bitcoin', to get the current bitcoin price
         "0x626974636f696e0000000000000000000000000000000000000000000000000000"
      ],
      "dataLength":"1",
      "requestId":"6",
      "requester":"0x8f108aab17e3b90f6855a73349511f5944b7e146"
   },
   "blockNumber":2182269,
   "transactionHash":
↪"0x673a1db9c675646c6319959f879bd8a3f711393667e6343c2125e707a70e8616",
}
12 : Response
{
   "args":{
      "data":"9204", // bitcoin price is 9204 USD at Fri Apr 27 00:44:58 EDT 2018
      "error":"0",
      "requestId":"6",
      "requester":"0x8f108aab17e3b90f6855a73349511f5944b7e146"
   },
```

(continues on next page)

(continued from previous page)

```
   "blockNumber":2182271,
   "transactionHash":
↪"0x1464d26cbab1238ce8ac4ac48cd2019425be59c451099d2437056ac6c253bf40",
}
```

CHAPTER 4

How to use various docker images

The preferred way to launch TC is via docker services. Of course, you'll need docker and docker-compose properly installed.

4.1 Setup

First, get the docker service files from

```
git clone https://github.com/bl4ck5un/Town-Crier-docker-sevices
cd rinkeby
```

The current version of TC uses Infura as a Web3 provider to access Rinkeby testnet. (Support for other networks as well as private nets are being added.) To use Infura, set the following environment variables (or put them in a `.env` file):

Warning: Keep `WEB3_INFURA_PROJECT_ID` and `WEB3_INFURA_API_SECRET` secrets.

4.2 Launch TC

Thanks to docker-compose, TC—the backend as well as the relay—can be launch together in one command: `docker-compose up`. To run it in the background, use `docker-compose up -d`. Other advanced uses of `docker-compose` can be found in their documentation.

There is a convenience Makefile that wraps around common docker-compose commands. For example, you can type `make up` instead of `docker-compose up`.

CHAPTER 5

Technical details of the `TownCrier.sol`

The `TownCrier` contract provides a uniform interface for queries from and replies to an application contract, which we also refer as a "Requester". This interface consists of the following three functions.

```
request(uint8 requestType, address callbackAddr, \
      bytes4 callbackFID, uint256 timestamp, \
      bytes32[] requestData) public payable returns(uint64);
```

An application contract sends queries to TC by calling function `request()`, and it needs to send the following parameters.

- `requestType`: indicates the query type. You can find the query types and respective formats that Town Crier currently supports on the Dev page.
- `callbackAddr`: the address of the application contract to which the response from Town Crier is forwarded.
- `callbackFID`: specifies the callback function in the application contract to receive the response from TC.
- `timestamp`: currently unused parameter. This parameter will be used in a future feature. Currently TC only responds to requests immediately. Eventually TC will support requests with a future query time pre-specified by `timestamp`. At present, developers can ignore this parameter and just set it to 0.
- `requestData`: data specifying query parameters. The format depends on the query type.

When the `request` function is called, a request is logged by event `RequestInfo()`. The function returns a `requestId` that is uniquely assigned to this request. The application contract can use the `requestId` to check the response or status of a request in its logs. The Town Crier server watches events and processes a request once logged by `RequestInfo()`.

Requesters must prepay the gas cost incurred by the Town Crier server in relaying a response to the application contract. `msg.value` is the amount of wei a requester pays and is recorded as `Request.fee`.

```
deliver(uint64 requestId, bytes32 paramsHash, uing64 error, bytes32 respData) public;
```

After fetching data and generating the response for the request with `requestId`, TC sends a transaction calling function `deliver()`. `deliver()` verifies that the function call is made by SGX and that the hash of query parameters is correct. Then it calls the callback function of the application contract to transmit the response.

The response includes `error` and `respData`. If `error = 0`, the application contract request has been successfully processed and the application contract can then safely use `respData`. The fee paid by the application contract for the request is consumed by TC. If `error = 1`, the application contract request is invalid or cannot be found on the website. In this case, similarly, the fee is consumed by TC. If `error > 1`, then an error has occured in the Town Crier server. In this case, the fee is fully refunded but the transaction cost for requesting by the application contract won't be compensated.

```
cancel(uint64 requestId) public returns(bool);
```

A requester can cancel a request whose response has not yet been issued by calling function `cancel()`. `requestId` is required to specify the query. The fee paid by the Appliciation Contract is then refunded (minus processing costs, denoted as cancellation fee).

For more details, you can look at the source code of the contract [TownCrier.sol].

CHAPTER 6

Indices and tables

- genindex
- modindex
- search