Theta Decentralized Edge Computing Platform

Preliminaries

The Theta decentralized edge computation platform involves three main actors.

- The **Task Initiator**, which represents the “users” who use the edge computing platform to solve their computational tasks. The Task Initiator posts tasks for the Edge Nodes to download and solve. It is also responsible for registering the tasks on the blockchain and providing the TFUEL rewards (or another token/cryptocurrency) for each task. The tasks can be anything ranging from solving a set of equations, finding novel protein structures to help fight COVID-19, to transcoding a video, and thousands of other applications that can leverage a network of distributed edge computing devices
- The **Edge Nodes**, which poll the Task Initiator to get the tasks. An Edge Node is a generic computational platform which can host various software including the solver for the tasks issued by the Task Initiator. Once a task is solved by an Edge Node, it can upload the solution.
- The **Smart Contracts** hosted by the blockchain. One of the smart contracts acts as a trustless escrow for the task rewards. Once a submitted solution is verified, the reward will be transferred to the solver (i.e. an Edge Node) automatically and transparently.

On-chain Solution Verification

When the solution size is small (i.e. a few kilobytes long), verifying the solution and rewarding the solver can be performed on-chain in a completely trustless fashion.

In some cases, the solution does not need to be kept in secret. For such cases, the plain-text solutions can be submitted to a blockchain smart contract directly and then gets verified on-chain.

There are also cases where the Task Initiator does not want to reveal solutions to the public. For such cases, we can ask the Edge Nodes to submit the encrypted solution to the blockchain. For the smart contract to verify the correctness of the encrypted solution without decrypting it, we propose to leverage the power of zero-knowledge proof techniques like zk-SNARK. An added benefit of zk-SNARK is to reduce the computational cost of solution validation. This is important since the cost of on-chain solution verification using smart contracts is proportional to the number of computational steps of the verification process. The zk-SNARK technique can magically turn any computation in the class *NP* into a verification process with a **constant** number of steps, which can always be conducted on-chain. Although this requires the Edge Nodes to generate the zk-SNARK proofs for the solutions, in many cases the computational overheads of proof-generation are manageable.

Below we provide the example Solidity smart contract for on-chain solution verification. The main contract `RewardPoolWithOnChainVerification` should be deployed by the Task Initiator. The contract has two functions/APIs:

- `registerTask()`: The function allows the Task Initiator to register a new task by providing the hash of the task, and the address of another smart contract `verifierContract`, which is responsible for verifying the solutions submitted for *this* task. The implementation of `verifierContract` should conform to the `VerifierInterface` interface in the code snippet. Note that the `verifierContract` for each individual task could be different. Before calling `registerTask()` to register a taks, the Task Initiator should deploy the `verifierContract` contract for that task on the blockchain and obtain its address. In addition, the Task Initiator should provide the token reward for solving this task. In the example code below, TFUEL is used as the reward token (via `msg.value`), but it can be any token/cryptocurrency in practice. After this function is called, the smart contract records this task on the blockchain.

- `submitSolution()`: The function allows an Edge Node to submit the solution to the smart contract, and get rewarded if the solution is valid. In the case where the Task Initiator allows the solutions to be published on-chain, the Edge Node can submit the plain-text solution as a byte string. In the case where the Task Initiator does not want to reveal the solutions, the Edge Node should submit the encrypted solution, and also provide the zk-SNARK proof `zkProof` (more details below). As described by the Solidity code, the `verifierContract` smart contract is then called to validate the correctness of the solution (via its `verifySolution()` function). If the solution passes the checks, the Edge Node is marked as the solver, and the TFUEL (or in the form of another token/cryptocurrency) reward will be sent to the solver automatically.

- **Encrypted Solution Handling**: If the Task Initiator does not want the plain-text solution to be available on the blockchain, it should publish its public key so that the Edge Node can encrypt the solution using the public key (e.g. via the ElGamal encryption protocol). The zk-SNARK proof `zkProof` should prove that:

    i. The plain-text solution solves the tasks (e.g. satisfies a set of constraints).

    ii. The solution submitted is indeed the encrypted plain-text solution using the public key of the Task Initiator.

  The `verifySolution()` function of the `verifierContract` smart contract should verify the correctness of the zk-SNARK proof.

- **Multiple Edge Nodes**: In the following code example, if multiple Edge Nodes solved the same task, only the first successfully submitted the solution to the smart contract can obtain the reward. As an extension, the implementation can be changed to allow multiple Edge Nodes to share the reward.

```
pragma solidity ^0.7.1;


interface VerifierInterface {

    function verifySolution(bytes memory solution, bytes memory zkProof) external pure returns (bool);

}

contract RewardPoolWithOnChainVerification {

    struct Task {
        bytes32 hash;
        uint    reward;
        address payable solver;

        // another smart contract which can verify the task solution on-chain
        address verifierContract;
    }

    event RegisterTask(bytes32 taskHash, uint rewardAmount);
    event VerifiedSolution(bytes32 taskHash, bytes zkProof, address solver);

    address public taskInitiator;
    mapping(bytes32 => Task) public taskMap;

    constructor() {
        taskInitiator = msg.sender;
    }

    function registerTask(bytes32 taskHash, address verifierContract) public payable {
        require(msg.sender != taskInitiator, "only task initiator can submit task hash");
        require(taskMap[taskHash].hash == bytes32(0x0), "the task is already registered");
        // Record the task on the blockchain
        taskMap[taskHash] = Task({
            hash: taskHash,
            reward: msg.value, // msg.value: amount of TFuelWei will be automatically transfer to the contract
            solver: address(0x0),
            verifierContract: verifierContract
        });

        emit RegisterTask(taskHash, msg.value);
    }

    function submitSolution(bytes32 taskHash, bytes memory solution, bytes memory zkProof) public {
        require(taskMap[taskHash].solver == address(0x0), "the task has been marked as solved");

        VerifierInterface verifier = VerifierInterface(taskMap[taskHash].verifierContract);
        if (verifier.verifySolution(solution, zkProof)) {
            address payable solver = msg.sender;
            taskMap[taskHash].solver = solver;

            uint reward = taskMap[taskHash].reward;
            taskMap[taskHash].solver.transfer(reward); // transfer the TFUEL reward to the solver

            emit VerifiedSolution(taskHash, zkProof, solver);
        }

    }

}
```

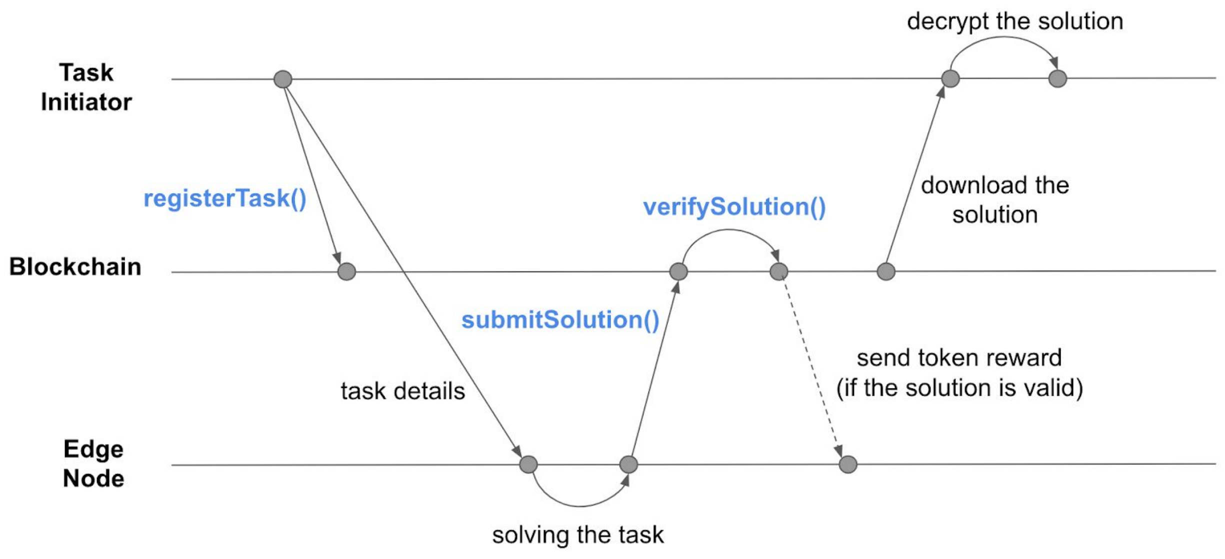
The flowchart below illustrates the interaction procedure among the three parties: the Task Initiator, the blockchain which hosts the smart contracts, and an Edge Node. Below is the interaction flow at a high level:

- **Step 1**: The Task Initiator creates a new task. Then it registers the task on the blockchain by calling `RewardPoolWithOnChainVerification.registerTask()`. Meanwhile, the Task Initiator can assign the Task to a designated Edge Node by sending the task details to the Edge Node through a secure channel.
    - Alternatively, the Edge Nodes may poll the `RewardPoolWithOnChainVerification` contract on regular intervals to see if any new task is available. If there are new tasks, an Edge Node can ping the Task Initiator to download the task details.
- **Step 2**: The Edge Node solves the task, and calls `RewardPoolWithOnChainVerification.submitSolution()` to submit the solution to the smart contract. In the case where the Task Initiator requires solution encryption, the Edge Node should also provide the zk-SNARK proof `zkProof`.
- **Step 3**: The `RewardPoolWithOnChainVerification.submitSolution()` function calls the `verifierContract` smart contract to verify the solution (and the

zk-SNARK proof if applicable). Upon successful verification, the `RewardPoolWithOnChainVerification` contract transfers the token reward to the Edge Node.

- **Step 4**. The Task Initiator downloads the verified encrypted solution from the blockchain, and uses its private key to decrypt the solution.

Note that the flowchart only depicts one Edge Node, but it can be extended to handle multiple Edge Nodes easily.



Off-chain Solution Verification

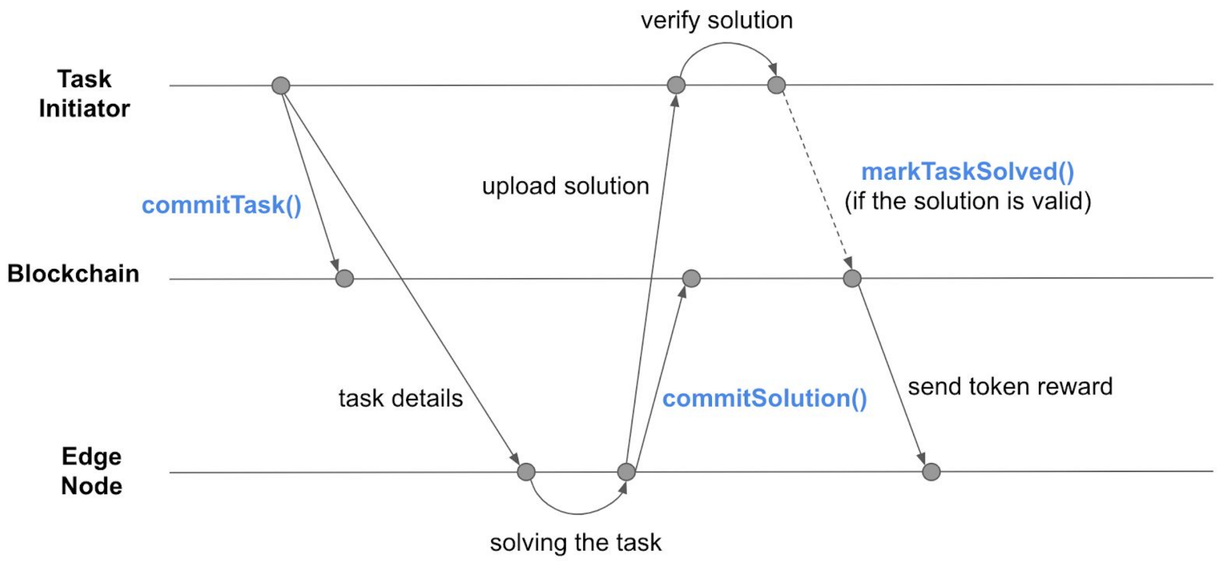
When the solution size is too large (e.g. a few megabytes or even larger), publishing the entire solution on chain might not be feasible. For such cases, the Task Initiator can verify the solutions off-chain and call the smart contracts to reward the Edge Nodes. Below is an example smart contract which handles the reward distribution with off-chain solution verification.

Compared to the on-chain verification case, after obtaining the solution to a task, an Edge Node only commits the hash of the solution on chain by calling the `RewardPoolWithOffChainVerification.commitSolution()` function, instead of uploading the solution to the blockchain. However, the Edge Node needs to send the complete solution to the Task Initiator through a secure channel. The Task Initiator then verifies the solution and marks the task as solved when it receives a valid solution.

Note that compared to the trustless on-chain verification, the off-chain solution verification flow requires a certain level of trust between the Edge Nodes and the Task Initiator. In particular, it requires the Task Initiator to call the `RewardPoolWithOffChainVerification.markTaskSolved()` function to transfer the token reward to the Edge Node that correctly solved the assigned task. An adversarial Task initiator could cheat on the Edge Nodes by skipping this step. In practice though, if this happens, the reputation of the Task Initiator will be tarnished quickly, and soon no Edge Node will solve tasks from this Task Initiator. A more advanced Task Initiator might attempt to change its on-chain identity by posting new `RewardPoolWithOffChainVerification` contracts from a different address. To guard against this attack, we can require the Task Initiator to deposit a certain number of non-redeemable collateral tokens to the `RewardPoolWithOffChainVerification` smart contract to begin with (see the `constructor()` function). This way, even though an adversarial Task Initiator can switch their on-chain identities by generating new pools, each pool-creation comes with non-negligible cost, which can effectively disincentivize malicious behaviors.

```
pragma solidity ^0.7.1;
pragma experimental ABIEncoderV2;

contract RewardPoolWithOffChainVerification {

    struct Task {
        bytes32 hash;
        uint    reward;
        address solver;
    }

    struct Solution {
        bytes32 taskHash;
        bytes32 solutionHash;
        address payable solver;
    }

    event CommitTask(bytes32 taskHash, uint rewardAmount);
    event CommitSolution(bytes32 taskHash, bytes32 solutionHash, address solver);
    event MarkSolutionAsSolved(bytes32 taskHash, bytes32 validSolutionHash, address solver);

    uint constant MIN_COLLATERAL = 10000000;

    address public taskInitiator;
    mapping(bytes32 => Task) public taskMap;
    mapping(bytes32 => Solution[]) public solutionMap;

    constructor() payable {
        taskInitiator = msg.sender;
        require(msg.value >= MIN_COLLATERAL); // to disincentivize malicious Task Initiators
    }

    function commitTask(bytes32 taskHash) public payable {
        require(msg.sender != taskInitiator, "only task initiator can submit task hash");
        require(taskMap[taskHash].hash == bytes32(0x0), "the task is already registered");

        // Record the task on the blockchain
        taskMap[taskHash] = Task({
            hash: taskHash,
            reward: msg.value, // msg.value amount of TFuelWei will be automatically transfer to the contract
            solver: address(0x0)
        });

        emit CommitTask(taskHash, msg.value);
    }

    function commitSolution(bytes32 taskHash, bytes32 solutionHash) public {
        require(taskMap[taskHash].solver == address(0x0), "the task has been marked as solved");

        solutionMap[taskHash].push(Solution({
            taskHash: taskHash,
            solutionHash: solutionHash,
            solver: msg.sender
        }));

        emit CommitSolution(taskHash, solutionHash, msg.sender);
    }

    function markTaskSolved(bytes32 taskHash, bytes32 validSolutionHash) public returns (bool) {
        require(msg.sender == taskInitiator, "only the task initiator can mark the task as solved");
        require(taskMap[taskHash].hash == taskHash, "incorrect task");
        require(taskMap[taskHash].solver == address(0x0), "the task has been marked as solved");

        Solution[] memory solutions = solutionMap[taskHash];
        for (uint i = 0; i < solutions.length; i++) {
            Solution memory solution = solutions[i];
            if (solution.solutionHash == validSolutionHash) {
                // found the first solver that committed the valid solution

                address solver = solution.solver;
                taskMap[taskHash].solver = solver; // mark the task as solved

                uint reward = taskMap[taskHash].reward;
                solution.solver.transfer(reward); // transfer the TFUEL reward to the solver

                emit MarkSolutionAsSolved(taskHash, validSolutionHash, solver);
                return true;
            }
        }

        return false;
    }

}
```

The interaction flowchart among the three partes is provided below. It is similar to the on-chain verification case and is pretty much self-explained.


verify solution
Task Initiator
markTaskSolved()
(if the solution is valid)
upload solution
commitTask()
Blockchain
task details
commitSolution()
send token reward
Edge Node
solving the task