The Loop of the Rings: A Fully Decentralized Cooperative System - Technical Report for Implementation

ANONYMOUS AUTHOR(S)

1 THE INITIATOR

For short, the Initiator decides whether the system should run the LoR or keep using the normal Ethereum transactions.

The constructor of the Initiator deploys a smart contract called *broadcast*. This contract handles all broadcast operations in the system. Namely using insert-only decentralized storage, only to let minor information be shared among all traders. Note that a broadcast operation does not require the cost of peer-to-peer connections. So, deploying a smart contract to handle the broadcast seemed to be the best choice.

The sign-up function deploys a trader smart contract. Counting the number of current users, it *broadcasts* that all users should switch to LoR.

Notice that the fields defined for the broadcast contract are all private. So, the necessary protection is provided.

```
pragma solidity >=0.8.2 <0.9.0;
import "./trader.sol";
import "./broadcast_sim.sol";

interface broadcast_interface{
    function switch_to_lor() external;
    function add_new_user_by_address(address adr, uint256 id) external;
}

contract initiator{
    mapping (uint256 => address) private traders;
    uint private randNonce = 0;
    bool private swtiched_to_lor;
    int private num_of_traders_signed_up;
    address private broadcast_addr;

    constructor() public{
        swtiched_to_lor = false;
        num_of_traders_signed_up = 0;
        broadcast_addr = address(new broadcast_sim());
    }

    function rand_id() private returns(uint256)
    {
        // increase nonce
        randNonce++;
        return uint256(sha256(abi.encodePacked(block.timestamp,msg.sender,randNonce)));
    }

    // The admin desires to signup a new trader
    function sign_up(int ara) external returns(address){
        if(swtiched_to_lor == true){
            return;
        }
        uint256 trader_id = rand_id();
        Trader trader = new Trader(trader_id, ara, broadcast_addr);
        num_of_traders_signed_up++;
        if(num_of_traders_signed_up >= 1000000){
            swtiched_to_lor = true;
            broadcast_interface(broadcast_addr).switch_to_lor();
        }
        broadcast_interface(broadcast_addr).add_new_user_by_address(address(trader), trader_id);
        traders[trader_id] = address(trader);
        return traders[trader_id];
    }

    function get_brroadcast_address() view external returns (address){
        return broadcast_addr;
    }
}
```

Fig. 1. The above figure shows that the initiator's corresponding smart contract handles the sign-up of the users. Also, it *broadcasts* the switch to LoR.

2 THE BROADCAST

From broadcasting the construction of invest/service coins and cooperation rings to the existence of a user (old and new), the broadcast's smart contract allows the system to perform the operations. Notice that there is no need to store anything more than the IDs/addresses of the elements in the storage of the broadcast. The *only* aim is to specify the existence of an entity and the address to find it. Observe that the entire storage of the broadcast's corresponding contract has the access level of *internal*. Namely, no other contract can modify them, unless via getter and setter functions that provide the necessary protection.

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.8.2 <0.9.0;

/**
 * @title broadcast_sim

 * @dev Store & retrieve value in a variable
 * @custom:dev-run-script ./scripts/deploy_with_ethers.ts
 */
contract broadcast_sim {

    mapping (uint256 => address) internal coins;
    mapping (uint256 => address) internal service_coin_table_map;
    uint256[] internal service_coin_table_ids;
    mapping(uint256 => address) internal users_addresses;
    uint256[] internal user_ids;
    mapping (uint256 => address) internal invest_coin_table_map;
    uint256[] internal invest_coin_table_ids;
    uint private randNonce = 0;
    mapping (uint256 => address) internal co_operation_table_map;
    mapping (uint256 => uint256[]) internal trader_id_to_co_op_ring_ids;
    uint256[] internal co_op_ring_ids;
    bool internal switched_to_lor = false;
```

Fig. 2. The above figure shows that the broadcast's corresponding smart contract only stores the IDs and addresses.

3 THE TRADER

The Trader's corresponding smart contract handles the main functionalities of the system. This smart contract keeps the receipts of the payments received and made. See Figure 4 and 5. Notice that each trader may provide and get services. It also has all functionalities required for a verification team member. It stores all fractal rings that the trader is a member of its verification team as well. Also, the trader provides the functions to check whether the users are satisfied at the end of each round. Moreover, the trader checks if the payments are made.

In order to present all *roles* the trader can get, there are several *interfaces* defined for the trader: Coin owner, Cooperation ring owner, and Verification team member. Since all of the information of a coin is stored by the trader, the Coin owner interface shows only the functionalities needed to get the allowed information from a coin. Note that the address of each trader is broadcasted in the system. So, for example, from the ID of a coin, it is easy to find the address of its owner. Then, it becomes easy to access the information of the coin. The implementation of the Coin table and the other tables can be seen in Figure 6.

Each trader stores the information of all fractal rings it once used to be a member of their verification teams. This makes it possible to verify if the vote the trader cast matched the majority's vote. See Figure 3.

```
mapping (uint256 => address[]) private ver_team_addresses_trader_was_a_member_of;
mapping (uint256 => uint256[]) private co_rings_of_frac_rings_received;
mapping (uint256 => bool) private votes_to_f_ids;

function get_vote_from_address(uint256[] memory co_ring_ids, address[] memory v_team_addrs, uint256 f_id) external returns (uint256) {
    if(broadcast(broadcast_addr).is_switched_to_lor() == false){
        return 0;
    }
    votes_to_f_ids[f_id] = false;
    ver_team_addresses_trader_was_a_member_of[f_id] = v_team_addrs;
    for(uint256 i = 0; i < co_ring_ids.length; i++ /*table in frac_ring_and_id[0]*/){
        CoOperationTable memory co_op = co_op_ring_owner(broadcast(broadcast_addr).get_co_ring_table(co_ring_ids[i])).get_co_op_ring(co_ring_ids[i]);
        uint256[] memory cids_binded_on = coin_owner(broadcast(broadcast_addr).get_coin_owner_address_by_id(co_op.trader_coin_id)).coin_ids_binded_on(co_op.trader_coin_id);
        for(uint256 j = 0; j < cids_binded_on.length; j++){
            if(compare(coin_owner(broadcast(broadcast_addr).get_coin_owner_address_by_id(cids_binded_on[j])).status_of(cids_binded_on[j]), "ready") ||
                coin_owner(broadcast(broadcast_addr).get_coin_owner_address_by_id(co_op.trader_coin_id)).get_ara_amount() < co_op.weight){
                return 0;
            }
        }
    }
    votes_to_f_ids[f_id] = true;
    return 1;
}
```

Fig. 3. The above figures show that the trader votes for a fractal ring's validity and stores the vote. Also, it keeps the IDs of other members of the verification team based on the ID of the corresponding fractal ring. Moreover, the IDs of the cooperation teams are stored.

```
contract Trader {
    uint256 private id;
    int private ara_amount;
    uint private randNonce = 0;
    uint private randNonceFrac = 0;
    address private broadcast_addr;
    mapping (uint256 => bool) private payment_received;
    mapping (uint256 => bool) private service_provided;
    mapping (uint256 => CoinTable) private service_coin_table_map;
    uint256[] private service_coin_table_ids;
    mapping (uint256 => bool) private payment_provided;
    mapping (uint256 => bool) private service_received;
    mapping (uint256 => CoinTable) private invest_coin_table_map;
    uint256[] private invest_coin_table_ids;
    mapping (uint256 => CoOperationTable) private co_op_rings_map;
    uint256[] private co_op_rings_ids;
```

Fig. 4. A portion of the crucial information each trader stores. Including the coins and cooperation rings generated. Also, the received payments and payments made.

4 HOW TO RUN THE SYSTEM

The easiest way to run our implementation of the LoR system is to deploy the Initiator's corresponding smart contract. To do so, one has to have an account and a *wallet* of a cryptocurrency. To deploy the smart contracts, we used the blockchain of Avalanche. Avalanche is an open-source platform for building decentralized applications. We encourage the readers to get more information from Avalanche's web page [1]. For development purposes, we used Avalanche's public API server[2] that allows developers to access the Avalanche network without having to run a node themselves. Avalanche includes three chains: The Exchange Chain (X-Chain), the Platform Chain (P-Chain), and the Contract Chain (C-Chain). The C-Chain is an implementation of the Ethereum Virtual Machine. The P-Chain is responsible for all validator and Subnet-level operations. Finally, the X-Chain is responsible for operations on digital smart assets known as Avalanche Native Tokens. For more information, please visit the corresponding web page[3]. In particular, we deployed our smart contracts on C-Chain by sending HTTP requests to Avalanche's public API. See Figure 7.

[1]https://docs.avax.network/overview/getting-started/avalanche-platform
[2]https://docs.avax.network/apis/avalanchego/public-api-server
[3]https://docs.avax.network/overview/getting-started/avalanche-platform

```
interface verification_team_member{
    function get_vote_from_address(uint256[] memory co_ring_ids, address[] memory f_ids, uint256 f_id) external returns (uint256);
    function submit_fractal_ring(uint256[] memory result, uint256 len_of_ver_tream, uint256 votes, uint256 f_id) external;
    function end_of_round_check(uint256 f_id) external returns (uint256);
    function payment_check(uint256 f_id) external returns (uint256);
}

interface co_op_ring_owner{
    function get_co_op_ring(uint256 g_id) view external returns(CoOperationTable memory);
}

interface coin_owner{
    function get_ara_amount() view external returns (int);
    function get_coin(uint256 cid, string memory coin_type) view external returns(CoinTable memory);
    function get_amount_b_o_invest_coin_by_id(uint256 cid) view external returns (int);
    function get_amount_b_o_service_coin_by_id(uint256 cid) view external returns (int);
    function set_bindings_invest_coin(uint256 coin_instance_id, uint256[] memory coin_ids_randomly_picked) external;
    function set_bindings_service_coin(uint256 coin_instance_id, uint256[] memory coin_ids_randomly_picked) external;
    function coin_ids_binded_on(uint256 cid) view external returns(uint256[] memory);
    function status_of(uint256 cid) view external returns (string memory);
    function service_received_or_not(uint256) view external returns (bool);
    function service_provided_or_not(uint256) view external returns (bool);
    function receive_service(uint256 result, uint256 c_id) external;
    function provide_service(uint256 c_id) external returns(uint256);
    function payment_received_or_not(uint256) view external returns (bool);
    function payment_provided_or_not(uint256) view external returns (bool);
    function pay(uint256 c_id) external;
    function receive_payment(uint256 c_id) external returns (bool);
}
```

Fig. 5. Different roles each trader may have. For each role, an interface is defined in the implementation.

```
struct CoinTable{
    uint256 coin_id;
    int amount_based_on_one_unit;
    string status;  // ready - blocked - expired/paid
    string type_of_coin;
    uint256 next_id_in_cooporation_ring;
    uint256 previous_id_in_cooperation_link;
    uint256 user_id_binded_on;
    uint256[] coin_ids_binded_on;
    uint256[] sha256_binded_on;
    // to submit a fractal ring using the same coin, or even the cases that such traders exist in a single co_op ring
    uint256 owner_id;
}

struct CoOperationTable{
    uint256 group_id;
    int number_of_group_members;
    int weight;
    uint256 next_id_in_fractal_ring;
    uint256 previous_id_in_fractal_ring;
    uint256 trader_coin_id;
    bytes32 trader_coin_sha256;
    int number_of_required_rounds;
    string co_status;
}

struct fractalRing{
    uint256 id;
    uint256[] coopRing_ids;
    uint256[] verification_team_ids;
}
```

Fig. 6. Different tables each trader stores. Including the Coin table, Cooperation ring table, and Fractal ring table.

Notice that performing the invocations of the functions, as well as deploying the smart contracts requires having *gas*. Gas is essential to the Ethereum network. It is the fuel that allows it to operate, in the same way that a car needs gasoline to run. It refers to the unit which measures the computational effort for executing operations on the Ethereum network[4]. So, if performing an operation makes an account run out of gas, the account owner has to buy more. Notice that a transaction may reach the *gas limit*. In this case, the user might need to increase the gas limit up to a certain value. Otherwise, everything should work smoothly and properly.

Once the Initiator's corresponding smart contract is deployed, the traders may be signed up via the invocation of the sign-up method. The system will automatically switch to LoR as soon as the number of users reaches one million. Each time a user gets signed up, the corresponding address of its smart contract will be returned. So, one can simulate the system by registering one million or more traders. Note that invoking the functions of each trader can be done by accessing its corresponding smart contract, using the address provided by the initiator. Once the system switches to LoR, one can deploy the smart contract of a trader and broadcast its address. Observe that the address of the broadcast's corresponding smart contract can be accessed via a method in the Initiator.

```python
    with open("compiled_code.json", "w") as file:
        json.dump(compiled_sol, file)
    # get bytecode
    bytecode = compiled_sol["contracts"][filename]["initiator"]["evm"][
        "bytecode"
    ]["object"]
    abi = json.loads(
        compiled_sol["contracts"][filename]["initiator"]["metadata"]
    )["output"]["abi"]
    w3 = Web3(Web3.HTTPProvider("https://avalanche-c-chain.publicnode.com"))
    w3.middleware_onion.inject(geth_poa_middleware, layer=0)
    chain_id = 43114
    my_address = "0xe72CfeDE68AF32d6354143DC35e61987BD384EEf"
    #    "0xCE6dD32848276F9C05134A3B9aD887EaFcBBb5bB"
    #  "0xCE6dD32848276F9C05134A3B9aD887EaFcBBb5bB"
    private_key = b'e6b2290a4b444f3d91945d02f9c7b267'
    #    b'c3e16d258afdcbf16e4af44e5353ff788c5dd58d67e51b65049631c0a1a697af'
    # 'e6b2290a4b444f3d91945d02f9c7b267'
    # initialize contract
    nonce = w3.eth.get_transaction_count(my_address)
    print("Deploying Contract!")
    ContactList = w3.eth.contract(abi=abi, bytecode=bytecode)
    transaction = ContactList.constructor().build_transaction(
        {
            "chainId": chain_id,
            "gasPrice": w3.eth.gas_price,
            "from": my_address,
            "nonce": nonce,
        }
    )
    sign_transaction = w3.eth.account.sign_transaction(transaction, private_key=private_key)
    transaction_hash = w3.eth.send_raw_transaction(sign_transaction.rawTransaction)
    transaction_receipt = w3.eth.wait_for_transaction_receipt(transaction_hash)
    contact_list = w3.eth.contract(address=transaction_receipt.contractAddress, abi=abi)
    return contact_list
```

Fig. 7. This figure shows how we deployed the smart contracts on the C-Chain of the Avalanche.

[4]https://mysl.nl/YBIr