/\*\*

   \* @title ContractName

   \* @dev ContractDescription

   \* @custom:dev-run-script file\_path

   \* @custom:dev-run-script NatSpec tag

   \*/

// SPDX-License-Identifier: MIT

pragma solidity 0.8.18;

contract ByzantineFaultTolerance {

    // Struct to represent a vote

    struct Vote {

        address voter;

        uint256 proposal;

    }

    // Enum to represent node roles

    enum NodeRole {Primary, Backup, Client}

    // Variables

    mapping(address => NodeRole) public nodes;

    mapping(uint256 => uint256) public proposalVotes;

    uint256 public currentProposal;

    uint256 public currentSequenceNumber;

    // Events

    event VoteReceived(address voter, uint256 proposal);

    event ConsensusReached(uint256 proposal);

    // Modifier to ensure only primary nodes can perform certain actions

    modifier onlyPrimary() {

        require(nodes[msg.sender] == NodeRole.Primary, "Only primary node can perform this action");

        \_;

    }

    // Modifier to ensure only backup nodes can perform certain actions

    modifier onlyBackup() {

        require(nodes[msg.sender] == NodeRole.Backup, "Only backup node can perform this action");

        \_;

    }

    // Constructor

    constructor() {

        // Set the roles of nodes

        nodes[msg.sender] = NodeRole.Primary;

        // Add backup nodes (in this example, there are 2 backup nodes)

        nodes[0x5B38Da6a701c568545dCfcB03FcB875f56beddC4] = NodeRole.Backup;

        nodes[0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2] = NodeRole.Backup;

    }

    // Client submits a vote

    function submitVote(uint256 proposal) external {

        require(nodes[msg.sender] == NodeRole.Client, "Only client can submit a vote");

        emit VoteReceived(msg.sender, proposal);

        // Broadcast the vote to backup nodes

        for (uint256 i = 0; i < 2; i++) {

            address backupNode = 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4; // Replace with actual backup node address

            // Send the vote to the backup node

            backupNode.call(abi.encodeWithSignature("receiveVote(address,uint256)", msg.sender, proposal));

        }

    }

    // Backup nodes receive votes

    function receiveVote(address voter, uint256 proposal) external onlyBackup {

        emit VoteReceived(voter, proposal);

        // Store the vote locally

        Vote memory vote = Vote(voter, proposal);

        uint256 voteHash = uint256(keccak256(abi.encode(voter, proposal)));

        proposalVotes[voteHash]++;

        // Check if consensus is reached

        if (proposalVotes[voteHash] >= 2) {

            emit ConsensusReached(proposal);

        }

    }

    // Primary node proposes a new proposal

    function propose(uint256 newProposal) external onlyPrimary {

        currentProposal = newProposal;

        currentSequenceNumber++;

        // Broadcast the proposal to backup nodes

        for (uint256 i = 0; i < 2; i++) {

            address backupNode = 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4; // Replace with actual backup node address

            // Send the proposal to the backup node

            backupNode.call(abi.encodeWithSignature("receiveProposal(uint256,uint256)", currentProposal, currentSequenceNumber));

        }

    }

    // Backup nodes receive proposals

    function receiveProposal(uint256 proposal, uint256 sequenceNumber) external onlyBackup {

        if (sequenceNumber > currentSequenceNumber) {

            currentProposal = proposal;

            currentSequenceNumber = sequenceNumber;

        }

    }

}