[Smart Contract Algorithms - Google Docs](https://docs.google.com/document/d/10Ccm4peg3PLPmA6Q-zpRkY5VBXOESScon-ijkhjTxk0/edit?tab=t.0)

[Smart Contracts Code](https://docs.google.com/document/d/1H0KfqCyPsxX256XTk8cqLSsXDnt5OAo1Gj2FQTK8Wew/edit?usp=sharing)

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.20;

import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

contract IoDRegistry is ReentrancyGuard {

// Structure definitions

struct BaseStation {

string pseudoId;

string publicKey;

bool isRegistered;

string[] publicParameters; // Added to store public parameters

}

struct User {

string pseudoId;

string publicKey;

bool isRegistered;

string accessCredential; // Added for access credentials

uint256 registrationTime;

}

struct Drone {

string pseudoId;

string publicKey;

bool isRegistered;

string droneType; // Added for drone details

string capabilities; // Added for drone capabilities

uint256 registrationTime;

bool isAvailable; // Availability status

string assignedUserId; // Current assigned user if any

}

struct GeoData {

string latitude;

string longitude;

uint256 timestamp;

string altitude; // Added for 3D positioning

string speed; // Added for monitoring speed

string batteryLevel; // Added for battery monitoring

}

struct Operation {

string operationId;

string userId;

string droneId;

uint256 startTime;

uint256 endTime;

bool isActive;

string operationType;

string operationStatus;

}

// Mappings

mapping(string => BaseStation) private baseStations;

mapping(string => User) private users;

mapping(string => Drone) private drones;

mapping(string => GeoData) private droneGeoData;

mapping(string => mapping(string => bytes32)) private sessionKeys; // Changed to map user-drone pairs to session keys

mapping(string => Operation[]) private userOperations;

mapping(string => string[]) private authorizedDronesForUser;

// Merkle Tree related variables

bytes32[] private droneHashes;

bytes32 public lastMerkleRoot;

mapping(bytes32 => bool) private verifiedMerkleNodes;

// System state variables

bool private systemInitialized = false;

uint256 public totalRegisteredDrones = 0;

uint256 public totalRegisteredUsers = 0;

uint256 public totalOperations = 0;

// Events

event SystemInitialized(string baseStationId, uint256 timestamp);

event BaseStationRegistered(string indexed pseudoId, string publicKey);

event UserRegistered(string indexed pseudoId, string publicKey);

event DroneRegistered(string indexed pseudoId, string publicKey, string droneType);

event DataPushed(string indexed droneId, string latitude, string longitude, uint256 timestamp);

event MerkleTreeCompleted(bytes32 root);

event SessionKeyGenerated(string indexed userId, string indexed droneId, bytes32 sessionKey);

event UserAuthenticated(string indexed userId, uint256 timestamp);

event DroneAuthenticated(string indexed droneId, uint256 timestamp);

event OperationStarted(string indexed operationId, string userId, string droneId, uint256 timestamp);

event OperationCompleted(string indexed operationId, uint256 timestamp);

event DroneAssigned(string indexed droneId, string indexed userId, uint256 timestamp);

event DroneReleased(string indexed droneId, string indexed userId, uint256 timestamp);

event EmergencyTriggered(string indexed droneId, uint256 timestamp, string emergencyType);

// Modifiers

modifier onlyInitialized() {

require(systemInitialized, "System not initialized.");

\_;

}

modifier onlyRegisteredBaseStation(string memory \_pseudoId) {

require(baseStations[\_pseudoId].isRegistered, "Base station not registered.");

\_;

}

modifier onlyRegisteredUser(string memory \_pseudoId) {

require(users[\_pseudoId].isRegistered, "User not registered.");

\_;

}

modifier onlyRegisteredDrone(string memory \_pseudoId) {

require(drones[\_pseudoId].isRegistered, "Drone not registered.");

\_;

}

modifier onlyAvailableDrone(string memory \_pseudoId) {

require(drones[\_pseudoId].isAvailable, "Drone not available.");

\_;

}

modifier onlyAuthorizedUser(string memory \_userId, string memory \_droneId) {

bool authorized = false;

for (uint i = 0; i < authorizedDronesForUser[\_userId].length; i++) {

if (keccak256(abi.encodePacked(authorizedDronesForUser[\_userId][i])) ==

keccak256(abi.encodePacked(\_droneId))) {

authorized = true;

break;

}

}

require(authorized, "User not authorized for this drone.");

\_;

}

// System Initialization phase

function initializeSystem(

string memory \_baseStationId,

string memory \_publicKey,

string[] memory \_publicParameters

) external {

require(!systemInitialized, "System already initialized.");

// Register the initial base station

baseStations[\_baseStationId] = BaseStation(

\_baseStationId,

\_publicKey,

true,

\_publicParameters

);

systemInitialized = true;

emit SystemInitialized(\_baseStationId, block.timestamp);

emit BaseStationRegistered(\_baseStationId, \_publicKey);

}

// Base Station functions

function registerBaseStation(

string memory \_pseudoId,

string memory \_publicKey,

string[] memory \_publicParameters

) external onlyInitialized {

require(!baseStations[\_pseudoId].isRegistered, "Base station already registered.");

baseStations[\_pseudoId] = BaseStation(\_pseudoId, \_publicKey, true, \_publicParameters);

emit BaseStationRegistered(\_pseudoId, \_publicKey);

}

function updateBaseStationParameters(

string memory \_pseudoId,

string[] memory \_newPublicParameters

) external onlyInitialized onlyRegisteredBaseStation(\_pseudoId) {

baseStations[\_pseudoId].publicParameters = \_newPublicParameters;

}

// User Registration phase

function registerUser(

string memory \_pseudoId,

string memory \_publicKey,

string memory \_accessCredential

) external onlyInitialized {

require(!users[\_pseudoId].isRegistered, "User already registered.");

users[\_pseudoId] = User(

\_pseudoId,

\_publicKey,

true,

\_accessCredential,

block.timestamp

);

totalRegisteredUsers++;

emit UserRegistered(\_pseudoId, \_publicKey);

}

// Drone Registration phase

function registerDrone(

string memory \_pseudoId,

string memory \_publicKey,

string memory \_droneType,

string memory \_capabilities

) external onlyInitialized {

require(!drones[\_pseudoId].isRegistered, "Drone already registered.");

drones[\_pseudoId] = Drone(

\_pseudoId,

\_publicKey,

true,

\_droneType,

\_capabilities,

block.timestamp,

true,

""

);

totalRegisteredDrones++;

// Add drone to Merkle Tree

droneHashes.push(keccak256(abi.encodePacked(\_pseudoId, \_publicKey, \_droneType)));

\_updateMerkleTree();

emit DroneRegistered(\_pseudoId, \_publicKey, \_droneType);

}

// Authentication & Session Key Generation phase

function authenticateUser(string memory \_pseudoId)

external

onlyInitialized

onlyRegisteredUser(\_pseudoId)

{

emit UserAuthenticated(\_pseudoId, block.timestamp);

}

function authenticateDrone(string memory \_pseudoId)

external

onlyInitialized

onlyRegisteredDrone(\_pseudoId)

{

emit DroneAuthenticated(\_pseudoId, block.timestamp);

}

function generateSessionKey(

string memory \_userId,

string memory \_droneId

)

external

onlyInitialized

onlyRegisteredUser(\_userId)

onlyRegisteredDrone(\_droneId)

nonReentrant

{

bytes32 sessionKey = keccak256(abi.encodePacked(

\_userId,

\_droneId,

block.timestamp,

users[\_userId].publicKey,

drones[\_droneId].publicKey

));

sessionKeys[\_userId][\_droneId] = sessionKey;

// Authorize user for this drone

bool alreadyAuthorized = false;

for (uint i = 0; i < authorizedDronesForUser[\_userId].length; i++) {

if (keccak256(abi.encodePacked(authorizedDronesForUser[\_userId][i])) ==

keccak256(abi.encodePacked(\_droneId))) {

alreadyAuthorized = true;

break;

}

}

if (!alreadyAuthorized) {

authorizedDronesForUser[\_userId].push(\_droneId);

}

emit SessionKeyGenerated(\_userId, \_droneId, sessionKey);

}

function getSessionKey(

string memory \_userId,

string memory \_droneId

)

external

view

onlyInitialized

onlyRegisteredUser(\_userId)

onlyRegisteredDrone(\_droneId)

returns (bytes32)

{

return sessionKeys[\_userId][\_droneId];

}

// Drone Operations

function assignDrone(

string memory \_userId,

string memory \_droneId,

string memory \_operationId,

string memory \_operationType

)

external

onlyInitialized

onlyRegisteredUser(\_userId)

onlyRegisteredDrone(\_droneId)

onlyAvailableDrone(\_droneId)

onlyAuthorizedUser(\_userId, \_droneId)

nonReentrant

{

// Mark drone as unavailable

drones[\_droneId].isAvailable = false;

drones[\_droneId].assignedUserId = \_userId;

// Create new operation

Operation memory newOperation = Operation(

\_operationId,

\_userId,

\_droneId,

block.timestamp,

0, // endTime will be set when operation completes

true,

\_operationType,

"IN\_PROGRESS"

);

userOperations[\_userId].push(newOperation);

totalOperations++;

emit DroneAssigned(\_droneId, \_userId, block.timestamp);

emit OperationStarted(\_operationId, \_userId, \_droneId, block.timestamp);

}

function releaseDrone(

string memory \_userId,

string memory \_droneId,

string memory \_operationId

)

external

onlyInitialized

onlyRegisteredUser(\_userId)

onlyRegisteredDrone(\_droneId)

onlyAuthorizedUser(\_userId, \_droneId)

nonReentrant

{

require(keccak256(abi.encodePacked(drones[\_droneId].assignedUserId)) ==

keccak256(abi.encodePacked(\_userId)), "Drone not assigned to this user.");

// Mark drone as available

drones[\_droneId].isAvailable = true;

drones[\_droneId].assignedUserId = "";

// Update operation status

bool operationFound = false;

for (uint i = 0; i < userOperations[\_userId].length; i++) {

if (keccak256(abi.encodePacked(userOperations[\_userId][i].operationId)) ==

keccak256(abi.encodePacked(\_operationId)) &&

userOperations[\_userId][i].isActive) {

userOperations[\_userId][i].isActive = false;

userOperations[\_userId][i].endTime = block.timestamp;

userOperations[\_userId][i].operationStatus = "COMPLETED";

operationFound = true;

break;

}

}

require(operationFound, "Active operation not found.");

emit DroneReleased(\_droneId, \_userId, block.timestamp);

emit OperationCompleted(\_operationId, block.timestamp);

}

// Emergency handling

function triggerEmergency(

string memory \_droneId,

string memory \_emergencyType

)

external

onlyInitialized

onlyRegisteredDrone(\_droneId)

nonReentrant

{

// Make drone available for emergency services

if (!drones[\_droneId].isAvailable) {

string memory previousUser = drones[\_droneId].assignedUserId;

// Mark drone as available for emergency

drones[\_droneId].isAvailable = true;

drones[\_droneId].assignedUserId = "";

emit DroneReleased(\_droneId, previousUser, block.timestamp);

}

emit EmergencyTriggered(\_droneId, block.timestamp, \_emergencyType);

}

// Location tracking

function updateDroneLocation(

string memory \_pseudoId,

string memory \_latitude,

string memory \_longitude,

string memory \_altitude,

string memory \_speed,

string memory \_batteryLevel

)

external

onlyInitialized

onlyRegisteredDrone(\_pseudoId)

nonReentrant

{

string memory trimmedId = \_trimString(\_pseudoId);

droneGeoData[trimmedId] = GeoData(

\_latitude,

\_longitude,

block.timestamp,

\_altitude,

\_speed,

\_batteryLevel

);

emit DataPushed(trimmedId, \_latitude, \_longitude, block.timestamp);

}

function getDroneData(string memory \_pseudoId)

external

view

onlyInitialized

onlyRegisteredDrone(\_pseudoId)

returns (

string memory latitude,

string memory longitude,

uint256 timestamp,

string memory altitude,

string memory speed,

string memory batteryLevel

)

{

string memory trimmedId = \_trimString(\_pseudoId);

require(droneGeoData[trimmedId].timestamp != 0, "No data available.");

GeoData memory data = droneGeoData[trimmedId];

return (

data.latitude,

data.longitude,

data.timestamp,

data.altitude,

data.speed,

data.batteryLevel

);

}

// Merkle Tree functions

function \_updateMerkleTree() internal {

if (droneHashes.length > 0) {

lastMerkleRoot = \_computeMerkleRoot(droneHashes);

emit MerkleTreeCompleted(lastMerkleRoot);

}

}

function \_computeMerkleRoot(bytes32[] memory \_hashes) private pure returns (bytes32) {

uint256 length = \_hashes.length;

if (length == 0) return bytes32(0);

if (length == 1) return \_hashes[0];

bytes32[] memory result = new bytes32[](\_hashes.length);

for (uint i = 0; i < \_hashes.length; i++) {

result[i] = \_hashes[i];

}

while (length > 1) {

uint256 i = 0;

uint256 j = 0;

while (i < length) {

if (i + 1 < length) {

result[j] = keccak256(abi.encodePacked(result[i], result[i + 1]));

} else {

result[j] = result[i];

}

i += 2;

j += 1;

}

length = j;

}

return result[0];

}

function verifyDroneInMerkleTree(

string memory \_droneId,

bytes32[] memory \_proof,

uint256 \_index

)

external

view

onlyInitialized

returns (bool)

{

require(drones[\_droneId].isRegistered, "Drone not registered");

bytes32 leaf = droneHashes[\_index];

bytes32 computedHash = leaf;

for (uint256 i = 0; i < \_proof.length; i++) {

bytes32 proofElement = \_proof[i];

if (\_index % 2 == 0) {

computedHash = keccak256(abi.encodePacked(computedHash, proofElement));

} else {

computedHash = keccak256(abi.encodePacked(proofElement, computedHash));

}

\_index = \_index / 2;

}

return computedHash == lastMerkleRoot;

}

// View functions

function isDroneRegistered(string memory \_pseudoId)

external

view

returns (bool)

{

return drones[\_pseudoId].isRegistered;

}

function isUserRegistered(string memory \_pseudoId)

external

view

returns (bool)

{

return users[\_pseudoId].isRegistered;

}

function isBaseStationRegistered(string memory \_pseudoId)

external

view

returns (bool)

{

return baseStations[\_pseudoId].isRegistered;

}

function getDroneDetails(string memory \_pseudoId)

external

view

onlyInitialized

onlyRegisteredDrone(\_pseudoId)

returns (

string memory publicKey,

string memory droneType,

string memory capabilities,

bool isAvailable,

string memory assignedUserId

)

{

Drone memory drone = drones[\_pseudoId];

return (

drone.publicKey,

drone.droneType,

drone.capabilities,

drone.isAvailable,

drone.assignedUserId

);

}

function getUserOperations(string memory \_userId)

external

view

onlyInitialized

onlyRegisteredUser(\_userId)

returns (Operation[] memory)

{

return userOperations[\_userId];

}

function getAuthorizedDrones(string memory \_userId)

external

view

onlyInitialized

onlyRegisteredUser(\_userId)

returns (string[] memory)

{

return authorizedDronesForUser[\_userId];

}

function getSystemStats()

external

view

returns (

bool initialized,

uint256 droneCount,

uint256 userCount,

uint256 operationCount,

bytes32 merkleRoot

)

{

return (

systemInitialized,

totalRegisteredDrones,

totalRegisteredUsers,

totalOperations,

lastMerkleRoot

);

}

// Utility function

function \_trimString(string memory str) internal pure returns (string memory) {

bytes memory strBytes = bytes(str);

uint256 start = 0;

uint256 end = strBytes.length;

while (start < end && (strBytes[start] == 0x20 || strBytes[start] == 0x09)) {

start++;

}

while (end > start && (strBytes[end - 1] == 0x20 || strBytes[end - 1] == 0x09)) {

end--;

}

bytes memory trimmed = new bytes(end - start);

for (uint256 i = start; i < end; i++) {

trimmed[i - start] = strBytes[i];

}

return string(trimmed);

}

}