GoodData Chain MainNet 1.0 Smart Contract Specs

Summary

To drive the whole flow via Smart Contract Events.

GoodData Service

workflow

1. `GDS` registers by `registerGDS` function with params (ip, port, /* public key, */ stake in).
2. `GDS` waiting for connection after `DO`/`MPC` registration.
3. If `GDS` wants to stop service, it will call `unregisterGDS` function.

structure

```
type GoodDataService struct {
    Address common.Address
    PublicKey string // Maybe needs it.
    DOs []DO
    MPCs []MPC
    StakeIn *big.Int
    IP string
    Port string
    Status GoodDataServiceStatus
}

type GoodDataServiceStatus int

const (
    GoodDataServiceStatusUnknown GoodDataServiceStatus = iota
    GoodDataServiceStatusInactive
    GoodDataServiceStatusActive
)
```

function list

1. Register

```
function registerGDS(ip, port, /* public key, */ stakeIn) {}
```

GDS to call. Smart Contract marks it as active. The public key is optional for authentication.

How to ensure the params, which are ip and port, are valid?

1. Unregister

```
function unregisterGDS() {}
```

GDS to call. Smart Contract marks it as inactive.

1. Available GDS (Read-Only)

function availableGDS() address->GDS {}

Smart Contract will return all available GDS .

1. GetGDS (Read-Only)

```
function getGDS(address) GDS {}
```

get all information of GDS by it's address

DO/MPC

workflow

Take DO for example:

1. DO registers by registerDO function with parameters(public key, local version, /*GDSAddr */, stake in, fees for each training) and emit DORegisterd event which

contains `GDS` address. Smart Contract stores all information of the `DO` , assigns a `GDS` to `DO` .

2. Once `GDS` monitor `DORegistered` event from chain, and param `GDSAddress` equal the it's own address, will wait for `DO` to communicate with it and use the public key of `DO` to decrypt private information.
3. Once `DO` monitors the `DORegistered` event from itself, try to communicate with `GDS` after getting information (ip, port) of `GDS` by the `getGDS` function and send private information of encryption by private key of `DO` .
4. `DO` send heartbeat information to the chain by `maintainHeartBeat` function every 3600 blocks.

structure

```
type DO struct {
    PublicKey string
    Version string
    StakeIn *big.Int
    FeesPerTraining *big.Int
    // MaxTrainingSlots = max(StakeIn / FeesPerTraining, minValue)
    MaxTrainingSlots uint64
    RunningQueries map[uuid]Query
    Status DOStatus
    LastUpdateBlockHeight uint64
    GDSAddress common.Address
}

var DOList map[address]DO

type DOStatus int

const (
    DOStatusUnknown DOStatus = iota
    DOStatusInactive
    DOStatusActive
)
```

function list

1. Register

```
  function registerDO(publicKey, localVersion, /* GDSAddr */, stakeIn,
feesPerTraining) {}
```

   DO/MPC to call

2. Unregister

```
  function unregisterDO() {}
```

   DO/MPC to call

3. UpdateDOStatus

```
  function updateDOStatus(uuid, doStatus) {}
```

   GDS calls this function, Smart Contract will switch DO status to doStatus.

4. GetDOStatus (Read-Only)

```
  function getDOStatus() returns (doStatus) {}
```

   getDOStatus returns DO status.

5. GetDOQueries

```
  function getDOQueries(uuid) returns ([]query) {}
```

6. MaintainHeartBeat

```
  function maintainHeartBeat(uuid, cpuUsage, gpuUsage, ramUsage)
```

   DO/MPC calls this function to maintain heartbeat.

Query

workflow

1. Once `QC` selects which DOs and MPCs to complete the query, `GDS` will estimate the fee (DO, MPC, GoodData combined) and send it back to `QC` .
2. `QC` ensures this order information (contains fee) from `GDS` . After the `QC` click `Next` button, `GDS` will generate and send the `query UUID` to it, and create the query by `createQuery` function which switches the status of query to 'pending'.
3. Once `QC` searches this order by `query UUID` in chain explorer or other service, confirms the order correct, it will submit the query to stake in the token, generate the `QuerySubmitted` event by `submitQuery` function and switch the status of query to 'submitted'.
    a. If the query type is vertical training, `MPC` will start to execute PSI. Completed PSI, `MPC` call `finishPSI` function, emit `PSIFinished` event and switch query status to 'psi finished' . `QC` call `confirmPSI` function after confirming PSI result, and emit `QueryDispatched` event.
    b. If the query type is horizontal training, emit a `QueryDispatched` event directly.
4. Once the query has maintained 'pending' status exceeding some block interval such as 3600 blocks, we think this query has failed, and Smart Contract rejects calling `submitQuery` function directly.
5. `submitQuery` function will check validity of the query (no timeout, enough fees, MPCs, DOs is active)
6. Once `DO/MPC` watches the `QueryDispatched` event on chain, it will start to train and connect to other nodes, and add logs by `addLog` function.

7. If one or more of the `MPCs` die which is recorded on chain or reported `DO` , `GDS` will assign new `MPCs` to training by `assignQueryToMPCs` function, which check validity (whether one or more of the `MPCs` die or not, and new `MPCs` is active). Died MPC will be fined for three rounds of aggregation fee.(see at Payment)
8. If `DO` dies, `QC` will get the full refund, `DO` that completed will get some rewards based on the round. Died `DO` will pay for the fee.
9. Once `DO/MPC` completes the query, it will call `updateDOQuery` function to notify `GDS` tasks completed. `GDS` monitor the status of the job of `DO/MPC` .
10. Once all done, Smart Contract will update the status of the query to 'completed', and generate a `QueryCompleted` event.
11. After the query is in the 'submitted' status, any unrecoverable error will switch the query's status to 'failed'.
12. After the query is in the 'submitted' status and `QC` canceled the query, Smart Contract will switch the query's status to 'canceled'. 60%

structure

```
type Query struct {
    UUID [16]byte // 16 byte, remove role identifier prefix
    QCUUID [16]byte // 16 byte, remove role identifier prefix
    QCAddress common.Address // it maybe need
    Type QueryType
    ParentQuery [16]byte
    // ModelID string
    // DataSetID string
    MPCs []MPC
    DOs []DO
    PSIDOIndex int
    Quote *big.Int // Quote >= RewardSchema.Fees
    Epoch uint64
    Batch uint64
    Status QueryStatus
    RewardSchema RewardSchema
    CreatedBlockNum uint64
}

type QueryType int

const (
    QueryTypeUnknown QueryType = iota
    QueryTypeVerticalTraining
    QueryTypeVerticalPrediction
    QueryTypeHorizontalTraining
    QueryTypeHorizontalPrediction
    QueryTypePSI
)

type QueryStatus int

const (
    QueryStatusUnknown QueryStatus = iota
    QueryStatusPending
    QueryStatusSubmitted
    QueryStatusDispatched
    QueryStatusCompleted
    QueryStatusFailed
    QueryStatusCanceled
)

type RewardSchema struct {
    ModelComplexityFactor int // default 1
    Fees *bit.Int
}
```

function list

1. CreateQuery

```
   // status -> pending
   function createQuery(uuid, qcUUID, psiUUID, type, /* modelID,
dataSetUUID, */ MPCs, DOs, Quote) {}
```

GDS to call

2. SubmitQuery

```
// status -> submitted | dispatched
function submitQuery(uuid, fees) {}
```

`QC` to call

3. CancelQuery

```
// status -> canceled
function cancelQuery(uuid) {}
```

`QC` can cancel the order by this function.

4. ConfirmPSI

```
// status -> dispatched
function confirmPSI(uuid) {}
```

`QC` calls this function, Smart Contract switch query status to 'dispatched' and validates the address of the QC which calls `submitQuery` and `confirmPSI` function are same.

5. UpdateQueryStatus

```
function updateQueryStatus(uuid, status) {}
```

`GDS` to call. We think `GDS` is honest now.

6. UpdateQueryProgress

```
function updateQueryProgress(uuid, epoch, batch)
```

`MPC` calls this function to update the epoch and batch of the query.

Implementation

Definition of Events and Functions

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.7.0;
pragma experimental ABIEncoderV2;

contract GoodDataContract {
    // const
    uint256 public constant MIN_STAKEIN_AMOUNT = 0.01 ether;
    uint256 public constant MIN_FEES_PER_TRAINING = 100 wei;
    uint32 public constant MAX_HEARTBEAT_INTERVAL = 3600;

    // heartbeat
    event Heartbeat(address nodeAddr, NodeType nodeType, uint8 cpuUsage, uint8 gpuUsage,
uint8 ramUsage);
    function maintainHeartbeat(NodeType nodeType, uint8 cpuUsage, uint8 gpuUsage, uint8
ramUsage) public {}

    function getHeartbeat(address nodeAddr, NodeType nodeType) public view returns(bool) {}

    function hasHeartbeat(address nodeAddr, NodeType nodeType) public returns(bool) {}

    // GDS
    function getGDS(address ssAddr) public view returns(GoodDataService memory) {}

    event GDSRegistered(address addr, uint32 ip, uint16 port, bytes publicKey);
    function registerGDS(uint32 ip, uint16 port, bytes memory publicKey) public payable {}

    event GDSUnregistered(address addr);
    function unregisterGDS() public {}

    // DO
    function getDO(address doAddr) public view returns(DO memory) {}

    event DORegistered(address _do, bytes publicKey, string localVersion, address
connectedGDS, uint256 stakeIn, uint256 feesPerTraining);
    function registerDO(
        bytes memory publicKey,
        string memory localVersion,
        uint256 feesPerTraining
    ) public payable {}

    // TODO:
    event ServerReady(bytes queryUUID, address nodeAddr, NodeType nodeType);
    function serverStartListening(bytes memory queryUUID, NodeType nodeType) public {}

    event DOUnregistered(address doAddr);
    function unregisterDO() public {}

    // MPC
    mapping (address => MPC) private mpcMaps;

    function getMPC(address mpcAddr) public view returns(MPC memory) {
        return mpcMaps[mpcAddr];
    }

    event MPCRegistered(address mpc, bytes publicKey, string localVersion, address
connectedGDS, uint256 stakeIn, uint256 feesPerTraining);
    function registerMPC(
        bytes memory publicKey,
        string memory localVersion,
        uint256 feesPerTraining
    ) public payable {}

    event PSIPlan(bytes queryUUID, bytes plan);
    function submitPSIPlan(bytes memory queryUUID, bytes memory plan) public {}

    event MPCUnregistered(address mpc);
    function unregisterMPC() public {}

    event LogAdded(bytes uuid, bytes payload);
    function addLog(bytes memory uuid, bytes memory payload) public virtual {}

    // Query
    // uuid => Query
    mapping (bytes => Query) private queryMaps;
    // uuid => ( address => TrainingProgress)
    mapping (bytes => mapping (address => TrainingProgress)) progressMaps;

    function getQuery(bytes memory uuid) public view returns(Query memory) {
        return queryMaps[uuid];
    }
    event QueryCreated(bytes uuid);
    function createQuery(
        bytes memory uuid,
        bytes memory qcUUID,
        QueryType queryType,
        bytes memory parentQuery,
        address [] memory mpcs,
        address [] memory dos,
        bytes [][] memory datasetUUIDs,
        uint256 quote
    ) public {}

    event QuerySubmitted(bytes uuid);
    event QueryDispatched(bytes uuid);
    function submitQuery(bytes memory uuid) public payable {}

    event QueryCanceled(bytes uuid);
    function cancelQuery(bytes memory uuid) public {}

    event QueryProgressUpdated(bytes uuid, uint64 epoch, uint64 totalEpoch, uint64 batchIndex,
uint64 totalBatchIndex);
    function updateQueryProgress(bytes memory uuid, uint64 epoch, uint64 totalEpoch, uint64
batchIndex, uint64 totalBatchIndex) public {}

    event QueryCompleted(bytes uuid);
    event QueryFailed(bytes uuid);

    function updateQueryStatus(bytes memory uuid, QueryStatus status, address [] memory
failedDOs, address [] memory failedMPCs) public {}
}

enum NodeType {
    Unknown,
    GoodDataService,
    DO,
    MPC
}

// GDS
struct GoodDataService {
    address addr;
    bytes publicKey;
    address [] dos;
    address [] mpcs;
    uint256 stakeIn;
    uint32 ip;
    uint16 port;
    GoodDataServiceStatus status;
}

enum GoodDataServiceStatus {
    Unregistered,
    Inactive,
    Active
}

// DO
struct DO {
    bytes publicKey;
    string version;
    uint256 stakeIn;
    uint256 feesPerTraining;
    uint64 maxTrainingSlots;
    bytes [] runningQueries;
    DOStatus status;
    uint256 lastUpdateBlockHeight;
    address connectedGDS;
}

enum DOStatus {
    Unregistered,
    Inactive,
    Active
}

// MPC
struct MPC {
    bytes publicKey;
    string version;
    uint256 stakeIn;
    uint256 feesPerTraining;
    uint64 maxTrainingSlots;
    bytes [] runningQueries;
    MPCStatus status;
    uint256 lastUpdateBlockHeight;
    address connectedGDS;
}
```

```
enum MPCStatus {
    Unregistered,
    Inactive,
    Active
}

// Query
struct Query {
    bytes uuid;
    bytes qcUUID;
    address qcAddress;
    QueryType typ;
    bytes parentQuery;
    address [] mpcs;
    address [] dos; // first DO is special for PSI
    bytes [][] datasetUUIDs;
    uint256 quote;
    // uint64 epoch;
    // uint64 totalEpoch;
    QueryStatus status;
    RewardSchema rewardSchema;
    uint256 createdBlockNum;
}

struct TrainingProgress {
    uint64 epoch;
    uint64 totalEpoch;
    uint64 batchIndex;
    uint64 totalBatchIndex;
}

enum QueryType {
    Unknown,
    VerticalTraining,
    VerticalPrediction,
    HorizontalTraining,
    HorizontalPrediction,
    PSI
}

enum QueryStatus {
    Unknown,
    Pending,
    Submitted,
    Dispatched,
    Completed,
    Failed,
    Canceled
}

struct RewardSchema {
    int256 ModelComplexityFactor;
    uint256 fees;
}
```