a)

**VM layer, VM GRPC server.**

For the storage of the State, Sparse Patricia Merkle Trie and MDBX database Serialisation.

The information stored in the leaf is:

h(h(h(contract\_hash,storage\_root),0),0)

Where:

contract\_hash is the hash of the contract’s definition.

storage\_root is the root of another Merkle-Patricia tree of height 251 that is constructed from the contract’s storage

h is the Pedersen hash function.

func (st \*state) GetContractState(address \*felt.Felt) (\*ContractState, error) {

leaf, err := st.stateTrie.Get(address)

if err != nil {

return nil, err

}

if leaf.Cmp(trie.EmptyNode.Bottom()) == 0 {

return &ContractState{new(felt.Felt), new(felt.Felt)}, nil

}

return st.manager.GetContractState(leaf)

}

**contractDefDb, err := db.NewMDBXDatabase(env, "CODE")**

**if err != nil {**

**t.Fail()**

**}**

**stateDb, err := db.NewMDBXDatabase(env, "STATE")**

func (m \*Manager) PutContractState(cs \*state.ContractState) error {

// Build protobuf struct

contractStatePB := &ContractState{

ContractHash: cs.ContractHash.ByteSlice(),

StorageRoot: cs.StorageRoot.ByteSlice(),

}

// Marshal to protobuf bytes

raw, err := proto.Marshal(contractStatePB)

if err != nil {

// Protobuf error

return err

}

// Put in database

return m.stateDatabase.Put(cs.Hash().ByteSlice(), raw)

}

b)**Database layer**

i)

database, err := **db.NewMDBXDatabase(env, "ABI")**

Contract ABI storage for cairo contracts deployed on starknet

// GetABI gets the ABI associated with the contract address. If the ABI does

// not exist, then returns nil without error.

func (m \*Manager) GetABI(contractAddress string) (\*Abi, error) {

// Build the key from contract address

key := []byte(contractAddress)

// Query to database

data, err := m.database.Get(key)

if err != nil {

return nil, err

}

// Unmarshal the data from database

abi := new(Abi)

if err := proto.Unmarshal(data, abi); err != nil {

return nil, err

}

return abi, nil

}

// PutABI puts the ABI to the contract address.

func (m \*Manager) PutABI(contractAddress string, abi \*Abi) error {

// Build the key from contract address

key := []byte(contractAddress)

value, err := proto.Marshal(abi)

if err != nil {

return err

}

if err := m.database.Put(key, value); err != nil {

return err

}

return nil

}

ii)database, err := **db.NewMDBXDatabase(env, "BLOCK")**

if err != nil {

t.Error(err)

}

Blockstore for starknet, design is very similar to tendermint block store which is used to implement side 2 chains.

/ GetBlockByHash search the block with the given block hash. If the block does

// not exist then returns nil. If any error happens, then panic.

func (manager \*Manager) GetBlockByHash(blockHash \*felt.Felt) (\*types.Block, error) {

// Build the hash key

hashKey := buildHashKey(blockHash)

// Search on the database

rawResult, err := manager.database.Get(hashKey)

if err != nil {

return nil, err

}

// Unmarshal the data

block, err := unmarshalBlock(rawResult)

if err != nil {

return nil, err

}

return block, nil

}

// GetBlockByNumber search the block with the given block number. If the block

// does not exist then returns nil. If any error happens, then panic.

func (manager \*Manager) GetBlockByNumber(blockNumber uint64) (\*types.Block, error) {

// Build the number key

numberKey := buildNumberKey(blockNumber)

// Search for the hash key

hashKey, err := manager.database.Get(numberKey)

if err != nil {

return nil, err

}

// Search for the block

rawResult, err := manager.database.Get(hashKey)

if err != nil {

return nil, err

}

// Unmarshal the data

block, err := unmarshalBlock(rawResult)

if err != nil {

return nil, err

}

return block, nil

}

3)**Contract layer**

stateDb, err := **db.NewMDBXDatabase(env, "STATE")**

if err != nil {

t.Fatal(err)

}

contractDefDb, err := **db.NewMDBXDatabase(env, "CONTRACT\_DEF")**

**if err != nil {**

t.Fatal(err)

}

func (m \*Manager) GetContractState(hash \*felt.Felt) (\*state.ContractState, error) {

raw, err := m.stateDatabase.Get(hash.ByteSlice())

if err != nil {

// Database error

return nil, err

}

// Unmarshal to protobuf struct

contractStatePB := &ContractState{}

err = proto.Unmarshal(raw, contractStatePB)

if err != nil {

// Protobuf error

return nil, err

}

// Build output struct

contractState := state.ContractState{

ContractHash: new(felt.Felt).SetBytes(contractStatePB.GetContractHash()),

StorageRoot: new(felt.Felt).SetBytes(contractStatePB.GetStorageRoot()),

}

return &contractState, nil

}

func (m \*Manager) PutContractState(cs \*state.ContractState) error {

// Build protobuf struct

contractStatePB := &ContractState{

ContractHash: cs.ContractHash.ByteSlice(),

StorageRoot: cs.StorageRoot.ByteSlice(),

}

// Marshal to protobuf bytes

raw, err := proto.Marshal(contractStatePB)

if err != nil {

// Protobuf error

return err

}

// Put in database

return m.stateDatabase.Put(cs.Hash().ByteSlice(), raw)

**4)Sync database**

**syncDatabase, err := db.NewMDBXDatabase(env, "SYNC")**

// StoreLatestBlockSaved stores the latest block sync.

func (m \*Manager) StoreLatestBlockSaved(latestBlockSaved int64) {

// Marshal the latest block sync

value, err := json.Marshal(latestBlockSaved)

if err != nil {

panic(any(fmt.Errorf("%w: %s", MarshalError, err)))

}

// Store the latest block sync

err = m.database.Put(latestBlockSavedKey, value)

if err != nil {

panic(any(fmt.Errorf("%w: %s", DbError, err.Error())))

}

}

// GetLatestBlockSaved returns the latest block sync.

func (m \*Manager) GetLatestBlockSaved() int64 {

// Query to database

data, err := m.database.Get(latestBlockSavedKey)

if err != nil {

// notest

if db.ErrNotFound == err {

return 0

}

panic(any(fmt.Errorf("%w: %s", DbError, err)))

}

if data == nil {

// notest

return 0

}

// Unmarshal the data from database

latestBlockSaved := new(int64)

if err := json.Unmarshal(data, latestBlockSaved); err != nil {

// notest

panic(any(fmt.Errorf("%w: %s", UnmarshalError, err.Error())))

}

return \*latestBlockSaved

}

// StoreLatestBlockSync stores the latest block sync.

func (m \*Manager) StoreLatestBlockSync(latestBlockSync int64) {

// Marshal the latest block sync

value, err := json.Marshal(latestBlockSync)

if err != nil {

panic(any(fmt.Errorf("%w: %s", MarshalError, err)))

}

// Store the latest block sync

err = m.database.Put(latestBlockSyncKey, value)

if err != nil {

panic(any(fmt.Errorf("%w: %s", DbError, err.Error())))

}

}

// GetLatestBlockSync returns the latest block sync.

func (m \*Manager) GetLatestBlockSync() int64 {

// Query to database

data, err := m.database.Get(latestBlockSyncKey)

if err != nil {

// notest

if errors.Is(err, db.ErrNotFound) {

return 0

}

panic(any(fmt.Errorf("%w: %s", DbError, err)))

}

// Unmarshal the data from database

latestBlockSync := new(int64)

if err := json.Unmarshal(data, latestBlockSync); err != nil {

// notest

panic(any(fmt.Errorf("%w: %s", UnmarshalError, err.Error())))

}

return \*latestBlockSync

}

// StoreLatestStateRoot stores the latest state root.

func (m \*Manager) StoreLatestStateRoot(stateRoot string) {

// Store the latest state root

err := m.database.Put(latestStateRoot, []byte(stateRoot))

if err != nil {

panic(any(fmt.Errorf("%w: %s", DbError, err.Error())))

}

}

// GetLatestStateRoot returns the latest state root.

func (m \*Manager) GetLatestStateRoot() string {

// Query to database

data, err := m.database.Get(latestStateRoot)

if err != nil {

// notest

if errors.Is(err, db.ErrNotFound) {

return ""

}

panic(any(fmt.Errorf("%w: %s", DbError, err)))

}

// Unmarshal the data from database

return string(data)

}

// StoreBlockOfProcessedEvent stores the block of the latest event processed,

func (m \*Manager) StoreBlockOfProcessedEvent(starknetFact, l1Block int64) {

key := []byte(fmt.Sprintf("%s%d", blockOfLatestEventProcessedKey, starknetFact))

// Marshal the latest block sync

value, err := json.Marshal(l1Block)

if err != nil {

panic(any(fmt.Errorf("%w: %s", MarshalError, err)))

}

// Store the latest block sync

err = m.database.Put(key, value)

if err != nil {

panic(any(fmt.Errorf("%w: %s", DbError, err.Error())))

}

}

// GetBlockOfProcessedEvent returns the block of the latest event processed,

func (m \*Manager) GetBlockOfProcessedEvent(starknetFact int64) int64 {

// Query to database

key := []byte(fmt.Sprintf("%s%d", blockOfLatestEventProcessedKey, starknetFact))

data, err := m.database.Get(key)

if err != nil {

// notest

if errors.Is(err, db.ErrNotFound) {

return 0

}

panic(any(fmt.Errorf("%w: %s", DbError, err)))

}

// Unmarshal the data from database

blockSync := new(int64)

if err := json.Unmarshal(data, blockSync); err != nil {

// notest

panic(any(fmt.Errorf("%w: %s", UnmarshalError, err.Error())))

}

return \*blockSync

}

5)**Transaction and receipt database.**

This is very similar to evm and evm db transaction receipts.

starktnet transaction state is serialised locally.

**txDb, err := db.NewMDBXDatabase(env, "TRANSACTION")**

**if err != nil {**

**t.Error(err)**

**}**

**receiptDb, err := db.NewMDBXDatabase(env, "RECEIPT")**

// GetTransaction searches in the database for the transaction associated with the

// given key. If the key does not exist then returns nil.

func (m \*Manager) **GetTransaction**(txHash \*felt.Felt) (types.IsTransaction, error) {

rawData, err := m.txDb.Get(txHash.ByteSlice())

if err != nil {

return nil, err

}

tx, err := unmarshalTransaction(rawData)

if err != nil {

return nil, err

}

return tx, nil

}

// PutReceipt stores new transactions receipts in the database. This method

// does not check if the key already exists. In the case, that the key already

// exists the value is overwritten.

func (m \*Manager) **PutReceipt(**txHash \*felt.Felt, txReceipt types.TxnReceipt) error {

rawData, err := marshalTransactionReceipt(txReceipt)

if err != nil {

return err

}

if err := m.receiptDb.Put(txHash.ByteSlice(), rawData); err != nil {

return err

}

return nil

}

// GetReceipt searches in the database for the transaction receipt associated

// with the given key. If the key does not exist then returns nil.

func (m \*Manager) **GetReceipt(**txHash \*felt.Felt) (types.TxnReceipt, error) {

rawData, err := m.receiptDb.Get(txHash.ByteSlice())

if err != nil {

return nil, err

}

receipt, err := unmarshalTransactionReceipt(rawData)

if err != nil {

return nil, err

}

return receipt, nil

}

6)Starknet RPC layer - fetching block information from block store/block manager.

func (s \*StarkNetRpc) **GetBlockWithTxHashes**(blockId \*BlockId) (any, error) {

b, err := getBlockById(blockId, s.blockManager, s.logger)

if err != nil {

return nil, err

}

return NewBlockWithTxHashes(b), nil

}

func (s \*StarkNetRpc) **GetBlockWithTxs**(blockId \*BlockId) (any, error) {

b, err := getBlockById(blockId, s.blockManager, s.logger)

if err != nil {

return nil, err

}

return NewBlockWithTxs(b, s.txnManager)

}

func (s \*StarkNetRpc) **GetStateUpdate**(blockId \*BlockId) (any, error) {

if blockId == nil {

return nil, nil

}

block, err := getBlockById(blockId, s.blockManager, s.logger)

if err != nil {

return nil, err

}

diff, err := s.synchronizer.GetStateDiff(block.BlockHash)

if err != nil {

return nil, err

}

return NewStateUpdate(diff), nil

}

func (s \*StarkNetRpc) **GetStorageAt**(address \*RpcFelt, key \*StorageKey, blockId \*BlockId) (any, error) {

b, err := getBlockById(blockId, s.blockManager, s.logger)

if err != nil {

return nil, err

}

\_state := state.New(s.stateManager, b.NewRoot)

value, err := \_state.GetSlot(address.Felt(), key.Felt())

if err != nil {

if errors.Is(err, db.ErrNotFound) {

return nil, ContractNotFound

}

s.logger.Errorw(err.Error(), "function", "GetStorageAt")

return nil, jsonrpc.NewInternalError(err.Error())

}

return value.Hex0x(), nil

}

func (s \*StarkNetRpc) **GetTransactionByHash**(transactionHash \*RpcFelt) (any, error) {

tx, err := s.txnManager.GetTransaction(transactionHash.Felt())

if err != nil {

if errors.Is(err, db.ErrNotFound) {

return nil, InvalidTxnHash

}

s.logger.Errorw(err.Error(), "function", "GetTransactionByHash")

return nil, jsonrpc.NewInternalError(err.Error())

}

return NewTxn(tx)

}

func (s \*StarkNetRpc) **GetTransactionByBlockIdAndIndex**(blockId \*BlockId, index \*uint64) (any, error) {

b, err := getBlockById(blockId, s.blockManager, s.logger)

if err != nil {

return nil, err

}

if index == nil || \*index >= b.TxCount {

return nil, InvalidTxnIndex

}

txHash := b.TxHashes[\*index]

tx, err := s.txnManager.GetTransaction(txHash)

if err != nil {

s.logger.Errorw(err.Error(), "function", "GetTransactionByBlockIdAndIndex")

return nil, jsonrpc.NewInternalError(err.Error())

}

return NewTxn(tx)

}

func (s \*StarkNetRpc) **GetTransactionReceipt**(transactionHash \*RpcFelt) (any, error) {

receipt, err := s.txnManager.GetReceipt(transactionHash.Felt())

if err != nil {

if errors.Is(err, db.ErrNotFound) {

return nil, InvalidTxnHash

}

s.logger.Errorw(err.Error(), "function", "GetTransactionReceipt")

return nil, jsonrpc.NewInternalError(err.Error())

}

return NewReceipt(receipt)

}

func (s \*StarkNetRpc) **GetClass**(classHash \*RpcFelt) (any, error) {

\_, latestBlockHash := s.synchronizer.LatestBlockSynced()

latestBlock, err := s.blockManager.GetBlockByHash(latestBlockHash)

if err != nil {

if errors.Is(err, db.ErrNotFound) {

return nil, InvalidContractClassHash

}

s.logger.Errorw(err.Error(), "function", "GetClass")

return nil, jsonrpc.NewInternalError(err.Error())

}

\_ = state.New(s.stateManager, latestBlock.NewRoot)

// TODO: implement class service

return nil, jsonrpc.NewInternalError("not implemented")

}

func (s \*StarkNetRpc) **GetClassHashAt**(blockId \*BlockId, address \*RpcFelt) (any, error) {

b, err := getBlockById(blockId, s.blockManager, s.logger)

if err != nil {

return nil, err

}

\_state := state.New(s.stateManager, b.NewRoot)

classHash, err := \_state.GetClassHash(address.Felt())

if err != nil {

if errors.Is(err, db.ErrNotFound) {

return nil, ContractNotFound

}

s.logger.Errorw(err.Error(), "function", "GetClassHashAt")

return nil, jsonrpc.NewInternalError(err.Error())

}

if classHash.IsZero() {

return nil, ContractNotFound

}

return classHash.Hex0x(), nil

}

func (s \*StarkNetRpc) **GetBlockTransactionCount**(blockId \*BlockId) (any, error) {

b, err := getBlockById(blockId, s.blockManager, s.logger)

if err != nil {

return nil, err

}

return b.TxCount, nil

}

func (s \*StarkNetRpc) **Call**(blockId \*BlockId, request \*FunctionCall) (any, error) {

b, err := getBlockById(blockId, s.blockManager, s.logger)

if err != nil {

return nil, err

}

if request == nil {

return nil, InvalidCallData

}

var (

callData = make([]\*felt.Felt, len(request.Calldata))

contractAddress \*felt.Felt

entryPointSelector \*felt.Felt

)

for i, data := range request.Calldata {

if !isFelt(data) {

return nil, InvalidCallData

}

callData[i] = new(felt.Felt).SetHex(data)

}

if !isFelt(request.ContractAddress) {

return nil, ContractNotFound

}

contractAddress = new(felt.Felt).SetHex(request.ContractAddress)

if !isFelt(request.EntryPointSelector) {

return nil, InvalidMessageSelector

}

entryPointSelector = new(felt.Felt).SetHex(request.EntryPointSelector)

\_state := state.New(s.stateManager, b.NewRoot)

out, err := s.vm.Call(

context.Background(),

\_state,

callData,

new(felt.Felt),

contractAddress,

entryPointSelector,

b.Sequencer,

)

if err != nil {

s.logger.Errorw(err.Error(), "function", "Call")

return nil, jsonrpc.NewInternalError(err.Error())

}

\_out := make([]string, len(out))

for i, v := range out {

\_out[i] = v.Hex0x()

}

return \_out, nil

}

// notest

func (s \*StarkNetRpc) **EstimateFee**(request \*InvokeTxn, blockId \*BlockId) (any, error) {

// TODO: implement

return nil, jsonrpc.NewInternalError("not implemented")

}

func (s \*StarkNetRpc) **BlockNumber()** (any, error) {

bNumber, \_ := s.synchronizer.LatestBlockSynced()

return bNumber, nil

7)**l1Client is the client that will be used to fetch the data that comes from the Ethereum Node.**

**client that will be used to fetch the data that comes from the Feeder Gateway.**

**Prometheus instrumentation of sync APIs.**

noOfRequests = promauto.NewCounterVec(prometheus.CounterOpts{

Name: "no\_of\_requests",

Help: "No. of requests sent to and received from the feeder gateway",

},

[]string{"Status", "Type"},

)

noOfABI = promauto.NewCounterVec(prometheus.CounterOpts{

Name: "no\_of\_abi",

Help: "Number of ABI requests sent to and received from the feeder gateway",

},

[]string{"Status"},

)

countStarknetSync = promauto.NewCounterVec(prometheus.CounterOpts{

Name: "count\_starknet\_sync",

Help: "Number of updates and commits made or failed",

},

**Feeder gateway retry mechanism** -

// retry mechanism for do requests

retryFuncForDoReq := func(req \*http.Request, httpClient HttpClient) (\*http.Response, error) {

var res \*http.Response

wait := 2 \* time.Second

for {

res, err = httpClient.Do(req)

if err != nil || res == nil || res.StatusCode != http.StatusOK {

wait \*= 2

Logger.With("Waiting:", wait.Seconds(), "Error", err).Info("Waiting to do again a request")

time.Sleep(wait)

continue

}

if res.StatusCode == http.StatusOK {

break

}

}

**Sample exponential back off algorithm that can be implemented for feeder gateway -**

**// BackoffPolicy implements a backoff policy, randomizing its delay and**

**// saturating at the final value in Millis**

type BackoffPolicy struct {

Millis []int

}

// Default is a backoff policy ranging up to 5 seconds

var Default = BackoffPolicy{

[]int{0, 10, 10, 100, 500, 500, 3000, 3000, 5000},

}

// Duration returns the time duration of the nth wait cycle in a

// backoff policy. This is b.Millis[n], randomized to avoid thundering herds

func (b BackoffPolicy) Duration(n int) time.Duration {

if n >= len(b.Millis) {

n = len(b.Millis) - 1

}

return time.Duration(jitter(b.Millis[n])) \* time.Millisecond

}

**//Exponential backoff structure**

**//Supports different methods for jitter implementation**

**//Full Jitter,Decorr Jitter are not implemented -> Equal jitter is demonstratred here**

**//Reference --> https://aws.amazon.com/blogs/architecture/exponential-backoff-and-jitter/**

**// jitter returns a random integer uniformly distributed in the range**

**// [0.5 \* millis .. 1.5 \* millis]**

func jitter(millis int) int {

if millis == 0 {

return 0

}

return millis / 2 \* rand.Intn(millis)

}

func init() {

rand.Seed(time.Now().UnixNano())

}

func doWork1() error {

return errors.New("hello error")

}

func main() {

**//For 1st attempt,2nd attempt,3rd attempt derive the value of jitter and sleep for that much amount of time**

**//Back Off policy to not stress the system and retries with jitter**

**State difference sync pipeline via starkware feeder gateway**

a)

var update \*feeder.StateUpdateResponse

var err error

update, err = a.client.GetStateUpdate("", strconv.FormatInt(latestStateDiffSynced, 10))

if err != nil {

a.logger.With("Error", err, "Block Number", latestStateDiffSynced).Info("Couldn't get state update")

continue

}

**//Channel for pushing state difference**

**a.buffer** <- fetchContractCode(stateUpdateResponseToStateDiff(\*update, latestStateDiffSynced), a.client, a.logger)

b)

var stateDiff types.StateUpdate

stateDiff.NewRoot = new(felt.Felt).SetHex(update.NewRoot)

stateDiff.BlockNumber = blockNumber

stateDiff.BlockHash = new(felt.Felt).SetHex(update.BlockHash)

stateDiff.OldRoot = new(felt.Felt).SetHex(update.OldRoot)

stateDiff.DeployedContracts = make([]types.DeployedContract, len(update.StateDiff.DeployedContracts))

for i, v := range update.StateDiff.DeployedContracts {

stateDiff.DeployedContracts[i] = types.DeployedContract{

Address: new(felt.Felt).SetHex(v.Address),

Hash: new(felt.Felt).SetHex(v.ContractHash),

}

}

stateDiff.StorageDiff = make(types.StorageDiff)

for contractAddress, memoryCells := range update.StateDiff.StorageDiffs {

kvs := make([]types.MemoryCell, 0)

for \_, cell := range memoryCells {

kvs = append(kvs, types.MemoryCell{

Address: new(felt.Felt).SetHex(cell.Key),

Value: new(felt.Felt).SetHex(cell.Value),

})

}

ca := new(felt.Felt).SetHex(contractAddress)

// Create felt and convert to string for consistency

stateDiff.StorageDiff[\*ca] = kvs

for different contract addresses

c)

**// fetchContractCode fetch the code of the contract from the Feeder Gateway.**

func fetchContractCode(stateDiff \*types.StateUpdate, client \*feeder.Client, logger \*zap.SugaredLogger) \*CollectorDiff {

collectedDiff := &CollectorDiff{

stateDiff: stateDiff,

Code: make(map[string]\*types.Contract, len(stateDiff.DeployedContracts)),

}

for \_, deployedContract := range stateDiff.DeployedContracts {

contractFromApi, err := client.GetFullContractRaw(deployedContract.Address.Hex0x(), "",

strconv.FormatInt(stateDiff.BlockNumber, 10))

if err != nil {

logger.With(

"Block Number", stateDiff.BlockNumber,

"Contract Address", deployedContract.Address.Hex0x(),

).Error("Error getting full contract")

return collectedDiff

}

contract := new(types.Contract)

err = contract.UnmarshalRaw(contractFromApi)

if err != nil {

logger.With(

"Block Number", stateDiff.BlockNumber,

"Contract Address", deployedContract.Address.Hex0x(),

).Error("Error unmarshalling contract")

}

collectedDiff.Code[deployedContract.Address.Hex0x()] = contract

}

return collectedDiff

}

d)

**type StateDiff struct {**

**//Storing KVS corresponding to contract address**

**StorageDiffs map[string][]KV `json:"storage\_diffs"`**

**DeployedContracts []DeployedContract `json:"deployed\_contracts"`**

**}**

e)

s.startingBlockNumber = s.syncManager.GetLatestBlockSync()

s.latestBlockNumberSynced = s.startingBlockNumber

// Get state

for collectedDiff := range s.stateDiffCollector.GetChannel() {

select {

case <-s.quit:

return nil

default:

start := time.Now()

err := s.updateState(collectedDiff)

{ **//Inserting state in Trie**

**// Storing latest fetched contract state in trie**

func (st \*state) SetSlots(address \*felt.Felt, slots []Slot) error {

contract, err := st.GetContractState(address)

if err != nil {

return err

}

storage := trie.New(st.manager, contract.StorageRoot, StorageTrieHeight)

for \_, slot := range slots {

if err := storage.Put(slot.Key, slot.Value); err != nil {

return err

}

}

contract.StorageRoot = storage.Root()

err = st.manager.PutContractState(contract)

if err != nil {

return err

}

return st.stateTrie.Put(address, contract.Hash())

}

if err != nil || s.state.Root().Cmp(collectedDiff.stateDiff.NewRoot) != 0 {

// In case some errors exist or the new root of the trie didn't match with

// the root we receive from the StateDiff, we have to revert the trie

s.logger.With("Error", err).Error("State update failed, reverting state")

prometheus.IncreaseCountStarknetStateFailed()

s.setStateToLatestRoot()

return err

}

prometheus.IncreaseCountStarknetStateSuccess()

prometheus.UpdateStarknetSyncTime(time.Since(start).Seconds())

s.syncManager.StoreLatestBlockSync(collectedDiff.stateDiff.BlockNumber)

**//Storing state difference in SYNC database**

if err := s.syncManager.StoreStateUpdate(collectedDiff.stateDiff, collectedDiff.stateDiff.BlockHash); err != nil {

return err

}

f)

**// StoreStateUpdate stores the state diff for the given block.**

func (m \*Manager) StoreStateUpdate(stateDiff \*types.StateUpdate, blockHash \*felt.Felt) error {

data, err := marshalStateUpdate(stateDiff)

if err != nil {

return err

}

return m.database.Put(stateDbKey(blockHash), data)

}

func (m \*Manager) GetStateUpdate(blockHash \*felt.Felt) (\*types.StateUpdate, error) {

data, err := m.database.Get(stateDbKey(blockHash))

if err != nil {

return nil, err

}

return unmarshalStateUpdate(data)

**Track sync status**

want := &types.SyncStatus{

StartingBlockHash: "0x0000000000000000000000000000000000000000000000000000000000000000",

StartingBlockNumber: fmt.Sprintf("%x", s.startingBlockNumber),

CurrentBlockHash: block.BlockHash.Hex0x(),

CurrentBlockNumber: fmt.Sprintf("%x", block.BlockNumber),

HighestBlockHash: s.stateDiffCollector.LatestBlock().BlockHash,

HighestBlockNumber: fmt.Sprintf("%x", s.stateDiffCollector.LatestBlock().BlockNumber),

}

Workflow if apisync is false and L1 collector is getting used -

**Smart contract definitions**

<https://github.com/starkware-libs/starkex-contracts/blob/master/evm-verifier/solidity/contracts/cpu/MemoryPageFactRegistry.sol>

<https://github.com/starkware-libs/starkex-contracts/blob/master/evm-verifier/solidity/contracts/gps/GpsStatementVerifier.sol>

**L1 collector pipeline**

**a)**

**func (l \*l1Collector) handleEvents() {**

// Get the block in which we are going to start processing events

// - 100 blocks back just in case.

initialBlock := l.manager.GetBlockOfProcessedEvent(l.latestBlockSynced) - int64(windowSize)

initialBlock = int64(math.Max(float64(initialBlock), float64(initialBlockForStarknetContract(l.chainID))))

blockNumber, err := l.l1Client.BlockNumber(context.Background())

if err != nil {

l.logger.Error("Error fetching latest block on Ethereum", "err", err)

return

}

// Keep updated the blockNumber of the latest block on L1

go func() {

blockNumber, err = l.l1Client.BlockNumber(context.Background())

if err != nil {

l.logger.Error("Error fetching latest block on Ethereum", "err", err)

}

time.Sleep(time.Second \* 5)

}()

// Process blocks until latest

for initialBlock < int64(blockNumber) {

if len(l.buffer) > 8 {

l.logger.Info("Buffer contains some elements, waiting to get more events")

time.Sleep(10 \* time.Second)

continue

}

err = l.processBatchOfEvents(initialBlock, int64(windowSize))

if err != nil {

l.logger.Error("Error processing batch of events", "err", err)

return

}

initialBlock += int64(windowSize)

}

// Subscribe for new blocks

**err = l.processSubscription(initialBlock)**

if err != nil {

l.logger.With("Error", err).Error("Error subscribing to events")

return

}

}

**b)**

// processSubscription iterates over the logs that has been thrown while subscribed to the L1

func (l \*l1Collector) **processSubscription(initialBlock int64)** error {

addresses := make([]common.Address, 0)

topics := make([]common.Hash, 0)

for k, v := range l.contractInfo {

addresses = append(addresses, k)

topics = append(topics, crypto.Keccak256Hash([]byte(v.Contract.Events[v.EventName].Sig)))

}

query := ethereum.FilterQuery{

FromBlock: big.NewInt(initialBlock),

Addresses: addresses,

}

hLog := make(chan types.Log)

sub, err := l.l1Client.SubscribeFilterLogs(context.Background(), query, hLog)

if err != nil {

l.logger.Info("Couldn't subscribe for incoming blocks")

return err

}

for {

select {

case err = <-sub.Err():

l.logger.With("Error", err).Debug("Error getting the latest logs")

case vLog := <-hLog:

event := map[string]interface{}{}

err = l.contractInfo[vLog.Address].Contract.UnpackIntoMap(event, l.contractInfo[vLog.Address].EventName, vLog.Data)

if err != nil {

l.logger.With("Error", err).Debug("Couldn't get event from log")

continue

}

eventChan := &types2.EventInfo{

Block: vLog.BlockNumber,

Event: event,

Address: l.contractInfo[vLog.Address].Address,

TxnHash: vLog.TxHash,

InitialBlockLogged: int64(vLog.BlockNumber),

}

l.processEvents(eventChan)

}

}

}

**c)**

func (l \*l1Collector) **updateLatestBlockOnChain()** {

go l.updatePendingBlock()

number, err := l.l1Client.BlockNumber(context.Background())

if err != nil {

l.logger.Error("Error subscribing to logs", "err", err)

return

}

// build query for the latest block

**query := ethereum.FilterQuery{**

**FromBlock: new(big.Int).SetInt64(int64(number) - int64(10\*windowSize)),**

**Addresses: []common.Address{l.starknetContractAddress},**

**Topics: [][]common.Hash{{crypto.Keccak256Hash([]byte(l.starknetABI.Events["LogStateUpdate"].Sig))}},**

**}**

logs, err := l.l1Client.FilterLogs(context.Background(), query)

if err != nil {

l.logger.Error("Error subscribing to logs", "err", err)

return

}

for \_, logFetched := range logs {

l.updateBlockOnChain(logFetched.Data)

}

subLogs := make(chan types.Log)

subscription, err := l.l1Client.SubscribeFilterLogs(context.Background(), query, subLogs)

defer subscription.Unsubscribe()

for logFetched := range subLogs {

l.updateBlockOnChain(logFetched.Data)

}

}

**// processPagesHashes takes an array of arrays of pages' hashes and**

**// converts them into memory pages by querying an ethereum client.**

**d)**

func (l \*l1Collector) **processPagesHashes**(pagesHashes [][32]byte, memoryContract ethAbi.ABI) ([][]\*big.Int, error) {

pages := make([][]\*big.Int, 0)

for \_, v := range pagesHashes {

// Get transactionsHash based on the memory page

hash := common.BytesToHash(v[:])

transactionHash, ok := l.memoryPageHash.Get(hash)

if !ok {

return nil, ErrorMemoryPageNotFound

}

txHash := transactionHash.(types2.TxnHash).Hash

txn, \_, err := l.l1Client.TransactionByHash(context.Background(), txHash)

if err != nil {

l.logger.With("Error", err, "Transaction Hash", v).

Error("Couldn't retrieve transactions")

return nil, err

}

// Parse Ethereum transaction calldata for Starknet transaction information

data := txn.Data()[4:] // Remove the method signature hash

inputs := make(map[string]interface{})

err = memoryContract.Methods["registerContinuousMemoryPage"].Inputs.UnpackIntoMap(inputs, data)

if err != nil {

l.logger.With("Error", err).Info("Couldn't unpack into map")

return nil, err

}

// Append calldata to pages

pages = append(pages, inputs["values"].([]\*big.Int))

}

return pages, nil

}

// parsePages converts an array of memory pages into a state diff that

// can be used to update the local state.

**// Deployed contracts**

**4, // Number of memory cells with deployed contract info**

**2, // Contract address**

**3, // Contract hash**

**1, // Number of constructor arguments**

**2, // Constructor argument**

**// State diffs**

**1, // Number of diffs**

**3, // Contract address**

**1, // Number of updates**

**3, // Key (Cairo memory address)**

**4, // Value**

**// Parsing deployed contracts information from memory page segments**

**// Parsing Memory page segments to determine deployed contracts, and populate KVs corresponding to contract address, to determine state diff**

func parsePages(pages [][]\*big.Int) \*types2.StateUpdate {

**// Remove first page**

pagesWithoutFirst := pages[1:]

// Flatter the pages recovered from Layer 1

pagesFlatter := make([]\*big.Int, 0)

for \_, page := range pagesWithoutFirst {

pagesFlatter = append(pagesFlatter, page...)

}

// Get the number of contracts deployed in this block

deployedContractsInfoLen := pagesFlatter[0].Int64()

pagesFlatter = pagesFlatter[1:]

deployedContracts := make([]types2.DeployedContract, 0)

**// Get the info of the deployed contracts**

deployedContractsData := pagesFlatter[:deployedContractsInfoLen]

// Iterate while contains contract data to be processed

for len(deployedContractsData) > 0 {

// Parse the Address of the contract

address := new(felt.Felt).SetBigInt(deployedContractsData[0])

deployedContractsData = deployedContractsData[1:]

// Parse the ContractInfo Hash

contractHash := new(felt.Felt).SetBigInt(deployedContractsData[0])

deployedContractsData = deployedContractsData[1:]

// Parse the number of Arguments the constructor contains

constructorArgumentsLen := deployedContractsData[0].Int64()

deployedContractsData = deployedContractsData[1:]

// Parse constructor arguments

constructorArguments := make([]\*felt.Felt, 0)

for i := int64(0); i < constructorArgumentsLen; i++ {

constructorArguments = append(constructorArguments, new(felt.Felt).SetBigInt(deployedContractsData[0]))

deployedContractsData = deployedContractsData[1:]

}

// Store deployed ContractInfo information

deployedContracts = append(deployedContracts, types2.DeployedContract{

Address: address,

Hash: contractHash,

ConstructorCallData: constructorArguments,

})

}

pagesFlatter = pagesFlatter[deployedContractsInfoLen:]

// Parse the number of contracts updates

numContractsUpdate := pagesFlatter[0].Int64()

pagesFlatter = pagesFlatter[1:]

storageDiff := make(types2.StorageDiff, 0)

**// Iterate over all the contracts that had been updated and collect the needed information**

for i := int64(0); i < numContractsUpdate; i++ {

// Parse the Address of the contract

address := new(felt.Felt).SetBigInt(pagesFlatter[0])

pagesFlatter = pagesFlatter[1:]

// Parse the number storage updates

numStorageUpdates := pagesFlatter[0].Int64()

pagesFlatter = pagesFlatter[1:]

kvs := make([]types2.MemoryCell, 0)

**for k := int64(0); k < numStorageUpdates; k++ {**

**kvs = append(kvs, types2.MemoryCell{**

**Address: new(felt.Felt).SetBigInt(pagesFlatter[0]),**

**Value: new(felt.Felt).SetBigInt(pagesFlatter[1]),**

**})**

**pagesFlatter = pagesFlatter[2:]**

**}**

storageDiff[\*address] = kvs

}

**//Storing KVs corresponding to contract address**

return &types2.StateUpdate{

DeployedContracts: deployedContracts,

StorageDiff: storageDiff,

}

}

**State difference is stored in database and updated in trie as described in pipeline for starkware feeder gateway API collector.**