Panther Design Problem Solution

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Brief points

Public blockchains have to maintain a certain amount of time lag between the production of adjacent blocks to ensure ample time for block propagation.

Also the block size should be small so as to ensure quick propagation of block through the network. Number of transactions in block need to be fairly limited.

1)Batch submission to mainchain - proof of panther blocks using Merkle root of blocks.

Steps -

1)A user deposits a cryptographic asset in the Panther contract on the mainchain.

2)Once deposited tokens get confirmed on the mainchain, token will appear on Panther chain using Panther Deposit Bridge.

3)Faster blocks approximately 1 second or less for almost negligible fees.

4)Whenever user wishes to withdraw they can withdraw tokens to the main ethereum by establishing proof of remaining tokens on root contract.

Ecosystem -

1)Dapp developers

2)End users

3)Stakers - need to deposit/stake tokens to qualify and play an important role in panther network. They validate the transaction and propose the checkpoints on the mainchain using proof of stake consensus mechanism with 2/3rd majority. They also choose block producers, among those who satisfy a criteria to produce blocks on sidechains.

4)Block producers choses by stakers who in turn enable faster blockchain generation times.

They have to provide a significant stake to be nominated.

Error cases in Deposit bridge –

1)Failed to map account address on panther side chain.

2)Snoops event – mint token on panther chain – transfer it to user account.

3)Deposit events processed in batches.

4)What if user is interested in other type of tokens say ERC721X, ERC721 for game purposes which require large batch transfers otherwise lot of gas will be consumed.

5)Withdraw event – transaction occurs on sidechain, a transaction receipt is generated.

Transaction gets submitted to mainnet with checkpoint.

6)bridge contract – if transaction evidence is present

Gets checkpoint receipt

7)Possibility of checkpoints getting stuck on the deposit bridge, leading to unclaimed checkpoints

1)Lot of deposit events get snooped by deposit bridge.

Batch process deposit events by Ethereum block numbers.

In case of error checkpoints become unclaimed , provision of storing unclaimed checkpoints and retrieving them in bridge.

When cumulative number of transactions in the block reach a certain threshold submit checkpoint to mainchain.

Checkpoints should not be submitted too soon or early to optimise gas fees.

Stake manager for block producers election.

Bridge block synchronization worker

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

type (

InitRequest = pctypes.BridgeCurrencyInitRequest

SubmitBlockToMainnetRequest = pctypes.SubmitBlockToMainnetRequest

SubmitBlockToMainnetResponse = pctypes.SubmitBlockToMainnetResponse

GetBlockRequest = pctypes.GetBlockRequest

GetBlockResponse = pctypes.GetBlockResponse

BridgeTxRequest = pctypes.BridgeTxRequest

BridgeTxResponse = pctypes.BridgeTxResponse

DepositRequest = pctypes.DepositRequest

BridgeTx = pctypes.BridgeTx

GetCurrentBlockResponse = pctypes.GetCurrentBlockResponse

GetCurrentBlockRequest = pctypes.GetCurrentBlockRequest

BridgeBookKeeping = pctypes.BridgeBookKeeping

BridgeBlock = pctypes.BridgeBlock

PendingTxs = pctypes.PendingTxs

CoinState = pctypes.BridgeCurrencyCoinState

Coin = pctypes.BridgeCurrencyCoin

Account = pctypes.BridgeCurrencyAccount

BalanceOfRequest = pctypes.BridgeCurrencyBalanceOfRequest

BalanceOfResponse = pctypes.BridgeCurrencyBalanceOfResponse

CoinResetRequest = pctypes.BridgeCurrencyCoinResetRequest

ExitCoinRequest = pctypes.BridgeCurrencyExitCoinRequest

WithdrawCoinRequest = pctypes.BridgeCurrencyWithdrawCoinRequest

TransferConfirmed = pctypes.BridgeCurrencyTransferConfirmed

GetBridgeTxRequest = pctypes.GetBridgeTxRequest

GetBridgeTxResponse = pctypes.GetBridgeTxResponse

GetUserSlotsRequest = pctypes.GetUserSlotsRequest

GetUserSlotsResponse = pctypes.GetUserSlotsResponse

UpdateBridgeAuthorityRequest = pctypes.BridgeCurrencyUpdateBridgeAuthorityRequest

GetPendingTxsRequest = pctypes.GetPendingTxsRequest

RequestBatchTally = pctypes.BridgeCurrencyRequestBatchTally

GetRequestBatchTallyRequest = pctypes.BridgeCurrencyGetRequestBatchTallyRequest

)

const (

CoinState\_DEPOSITED = pctypes.BridgeCurrencyCoinState\_DEPOSITED

CoinState\_EXITING = pctypes.BridgeCurrencyCoinState\_EXITING

CoinState\_CHALLENGED = pctypes.BridgeCurrencyCoinState\_CHALLENGED

CoinState\_EXITED = pctypes.BridgeCurrencyCoinState\_EXITED

contractBridgeCurrencyTransferConfirmedEventTopic = "event:BridgeCurrencyTransferConfirmed"

BridgeAuthorityRole = "pCurrency\_role\_BridgeAuthority"

addressMapperContractName = "addressmapper"

)

type BridgeCurrency struct {

}

var (

blockHeightKey = []byte("pCurrency\_height")

pendingTXsKey = []byte("pCurrency\_pending")

accountKeyPrefix = []byte("account")

BridgeMerkleTopic = "pCurrency\_mainnet\_merkle"

SubmitBlockConfirmedEventTopic = "pCurrency:submitblockconfirmed"

ExitConfirmedEventTopic = "pCurrency:exitconfirmed"

WithdrawConfirmedEventTopic = "pCurrency:withdrawconfirmed"

ResetConfirmedEventTopic = "pCurrency:resetconfirmed"

DepositConfirmedEventTopic = "pCurrency:depositconfirmed"

ChangeBridgeAuthorityPermission = []byte("change\_BridgeAuthority")

SubmitEventsPermission = []byte("submit\_events")

ErrNotAuthorized = errors.New("sender is not authorized to call this method")

)

func accountKey(addr pantherBridge.Address) []byte {

return util.PrefixKey(accountKeyPrefix, addr.Bytes())

}

func coinKey(slot uint64) []byte {

var buf bytes.Buffer

binary.Write(&buf, binary.BigEndian, slot)

return util.PrefixKey([]byte("coin"), buf.Bytes())

}

func blockKey(height common.BigUInt) []byte {

return util.PrefixKey([]byte("pCurrency\_block\_"), []byte(height.String()))

}

func requestBatchTallyKey() []byte {

return []byte("request\_batch\_tally")

}

func (c \*BridgeCurrency) Meta() (plugin.Meta, error) {

return plugin.Meta{

Name: "BridgeCurrency",

Version: "1.0.0",

}, nil

}

func (c \*BridgeCurrency) GetRequestBatchTally(ctx contract.StaticContext, req \*GetRequestBatchTallyRequest) (\*RequestBatchTally, error) {

tally := &RequestBatchTally{}

if err := ctx.Get(requestBatchTallyKey(), tally); err != nil {

if err == contract.ErrNotFound {

return tally, nil

}

return nil, errors.Wrapf(err, "error while getting request batch tally")

}

return tally, nil

}

func (c \*BridgeCurrency) GetPendingTxs(ctx contract.StaticContext, req \*GetPendingTxsRequest) (\*PendingTxs, error) {

pending := &PendingTxs{}

if err := ctx.Get(pendingTXsKey, pending); err != nil {

// If this key does not exists, that means contract hasnt executed

// any submit block request. We should return empty object in that

// case.

if err == contract.ErrNotFound {

return pending, nil

}

return nil, errors.Wrapf(err, "error while getting pendingTXsKey")

}

return pending, nil

}

func (c \*BridgeCurrency) registerBridgeAuthority(ctx contract.Context, pbBridgeAuthority \*types.Address, currentBridgeAuthority \*pantherBridge.Address) error {

if pbBridgeAuthority == nil {

return fmt.Errorf("BridgeAuthority address cannot be null")

}

newBridgeAuthorityAddr := pantherBridge.UnmarshalAddressPB(pbBridgeAuthority)

if newBridgeAuthorityAddr.IsEmpty() {

return fmt.Errorf("BridgeAuthority address cannot be empty")

}

// Revoke/Grant all permission as it is single BridgeAuthority atm

if currentBridgeAuthority != nil {

ctx.RevokePermissionFrom(\*currentBridgeAuthority, ChangeBridgeAuthorityPermission, BridgeAuthorityRole)

ctx.RevokePermissionFrom(\*currentBridgeAuthority, SubmitEventsPermission, BridgeAuthorityRole)

}

ctx.GrantPermissionTo(newBridgeAuthorityAddr, ChangeBridgeAuthorityPermission, BridgeAuthorityRole)

ctx.GrantPermissionTo(newBridgeAuthorityAddr, SubmitEventsPermission, BridgeAuthorityRole)

return nil

}

func (c \*BridgeCurrency) Init(ctx contract.Context, req \*InitRequest) error {

//params := req.Params

if err := c.registerBridgeAuthority(ctx, req.BridgeAuthority, nil); err != nil {

return errors.Wrapf(err, "unable to register new BridgeAuthority")

}

ctx.Set(blockHeightKey, &BridgeBookKeeping{CurrentHeight: &types.BigUInt{

Value: \*pantherBridge.NewBigUIntFromInt(0),

}})

return nil

}

func round(num, near int64) int64 {

if num == 0 {

return near

}

if num%near == 0 { //we always want next value

return num + near

}

return ((num + (near - 1)) / near) \* near

}

func (c \*BridgeCurrency) UpdateBridgeAuthority(ctx contract.Context, req \*UpdateBridgeAuthorityRequest) error {

if hasPermission, \_ := ctx.HasPermission(ChangeBridgeAuthorityPermission, []string{BridgeAuthorityRole}); !hasPermission {

return ErrNotAuthorized

}

currentBridgeAuthority := ctx.Message().Sender

return c.registerBridgeAuthority(ctx, req.NewBridgeAuthority, &currentBridgeAuthority)

}

func (c \*BridgeCurrency) emitSubmitBlockConfirmedEvent(ctx contract.Context, numPendingTransactions int, blockHeight \*types.BigUInt, merkleHash []byte) error {

marshalled, err := proto.Marshal(&pctypes.BridgeCurrencySubmitBlockConfirmedEvent{

NumberOfPendingTransactions: uint64(numPendingTransactions),

CurrentBlockHeight: blockHeight,

MerkleHash: merkleHash,

})

if err != nil {

return err

}

ctx.EmitTopics(marshalled, SubmitBlockConfirmedEventTopic)

return nil

}

func (c \*BridgeCurrency) emitExitConfirmedEvent(ctx contract.Context, owner \*types.Address, slot uint64) error {

marshalled, err := proto.Marshal(&pctypes.BridgeCurrencyExitConfirmedEvent{

Owner: owner,

Slot: slot,

})

if err != nil {

return err

}

ctx.EmitTopics(marshalled, ExitConfirmedEventTopic)

return nil

}

func (c \*BridgeCurrency) emitWithdrawConfirmedEvent(ctx contract.Context, coin \*Coin, owner \*types.Address, slot uint64) error {

marshalled, err := proto.Marshal(&pctypes.BridgeCurrencyWithdrawConfirmedEvent{

Coin: coin,

Owner: owner,

Slot: slot,

})

if err != nil {

return err

}

ctx.EmitTopics(marshalled, WithdrawConfirmedEventTopic)

return nil

}

func (c \*BridgeCurrency) emitResetConfirmedEvent(ctx contract.Context, owner \*types.Address, slot uint64) error {

marshalled, err := proto.Marshal(&pctypes.BridgeCurrencyResetConfirmedEvent{

Owner: owner,

Slot: slot,

})

if err != nil {

return err

}

ctx.EmitTopics(marshalled, ResetConfirmedEventTopic)

return nil

}

func (c \*BridgeCurrency) emitDepositConfirmedEvent(ctx contract.Context, coin \*Coin, owner \*types.Address) error {

marshalled, err := proto.Marshal(&pctypes.BridgeCurrencyDepositConfirmedEvent{

Coin: coin,

Owner: owner,

})

if err != nil {

return err

}

ctx.EmitTopics(marshalled, DepositConfirmedEventTopic)

return nil

}

func (c \*BridgeCurrency) SubmitBlockToMainnet(ctx contract.Context, req \*SubmitBlockToMainnetRequest) (\*SubmitBlockToMainnetResponse, error) {

// prevent this being called to often

//if we have a half open block we should flush it

//Raise blockheight

if hasPermission, \_ := ctx.HasPermission(SubmitEventsPermission, []string{BridgeAuthorityRole}); !hasPermission {

return nil, ErrNotAuthorized

}

pbk := &BridgeBookKeeping{}

ctx.Get(blockHeightKey, pbk)

pending := &PendingTxs{}

ctx.Get(pendingTXsKey, pending)

leaves := make(map[uint64][]byte)

if len(pending.Transactions) == 0 {

ctx.Logger().Warn("No pending transaction, returning")

return &SubmitBlockToMainnetResponse{}, nil

} else {

ctx.Logger().Warn("Pending transactions, raising blockheight")

// do this rounding in a bigint safe way

// round to nearest 1000

roundedInt := round(pbk.CurrentHeight.Value.Int64(), 1000)

pbk.CurrentHeight.Value = \*pantherBridge.NewBigUIntFromInt(roundedInt)

ctx.Set(blockHeightKey, pbk)

}

for \_, v := range pending.Transactions {

if v.PreviousBlock == nil || v.PreviousBlock.Value.Int64() == int64(0) {

hash, err := soliditySha3(v.Slot)

if err != nil {

return nil, err

}

v.MerkleHash = hash

} else {

hash, err := rlpEncodeWithSha3(v)

if err != nil {

return nil, err

}

v.MerkleHash = hash

}

leaves[v.Slot] = v.MerkleHash

}

smt, err := mamamerkle.NewSparseMerkleTree(64, leaves)

if err != nil {

return nil, err

}

for \_, v := range pending.Transactions {

v.Proof = smt.CreateMerkleProof(v.Slot)

}

merkleHash := smt.Root()

pb := &BridgeBlock{

MerkleHash: merkleHash,

Transactions: pending.Transactions,

Uid: pbk.CurrentHeight,

}

err = ctx.Set(blockKey(pbk.CurrentHeight.Value), pb)

if err != nil {

return nil, err

}

ctx.EmitTopics(merkleHash, BridgeMerkleTopic)

//Clear out old pending transactions

err = ctx.Set(pendingTXsKey, &PendingTxs{})

if err != nil {

return nil, err

}

c.emitSubmitBlockConfirmedEvent(ctx, len(pending.Transactions), pbk.CurrentHeight, merkleHash)

return &SubmitBlockToMainnetResponse{MerkleHash: merkleHash}, nil

}

func (c \*BridgeCurrency) verifyBridgeRequest(ctx contract.Context, req \*BridgeTxRequest) error {

if req.Bridgetx == nil || req.Bridgetx.Sender == nil || req.Bridgetx.Denomination == nil ||

req.Bridgetx.PreviousBlock == nil || req.Bridgetx.NewOwner == nil {

return fmt.Errorf("one or more required fields are nil")

}

claimedSender := pantherBridge.UnmarshalAddressPB(req.Bridgetx.Sender)

pantherBridgeTx := &Bridge\_Currency.PantherBridgeTx{

Slot: req.Bridgetx.Slot,

Denomination: req.Bridgetx.Denomination.Value.Int,

Owner: ethcommon.BytesToAddress(req.Bridgetx.NewOwner.Local),

PrevBlock: req.Bridgetx.PreviousBlock.Value.Int,

TXProof: req.Bridgetx.Proof,

}

calculatedBridgeTxHash, err := pantherBridgeTx.Hash()

if err != nil {

return errors.Wrapf(err, "unable to calculate BridgeTx hash")

}

senderEthAddressFromBridgeSig, err := evmcompat.RecoverAddressFromTypedSig(

calculatedBridgeTxHash, req.Bridgetx.Signature, []evmcompat.SignatureType{

evmcompat.SignatureType\_EIP712,

evmcompat.SignatureType\_GETH,

evmcompat.SignatureType\_TREZOR,

},

)

if err != nil {

return errors.Wrapf(err, "unable to recover sender address from Bridgetx signature")

}

addressMapper, err := ctx.Resolve(addressMapperContractName)

if err != nil {

return errors.Wrapf(err, "error while resolving address mapper contract address")

}

addressMapperResponse := &amtypes.AddressMapperGetMappingResponse{}

if err := contract.StaticCallMethod(ctx, addressMapper, "GetMapping", &amtypes.AddressMapperGetMappingRequest{

From: ctx.Message().Sender.MarshalPB(),

}, addressMapperResponse); err != nil {

return errors.Wrapf(err, "error while getting mapping from address mapper contract.")

}

if bytes.Compare(senderEthAddressFromBridgeSig.Bytes(), claimedSender.Local) != 0 ||

bytes.Compare(claimedSender.Local, addressMapperResponse.To.Local) != 0 {

return fmt.Errorf("Bridgetx signature doesn't match sender")

}

return nil

}

func (c \*BridgeCurrency) BridgeTxRequest(ctx contract.Context, req \*BridgeTxRequest) error {

if err := c.verifyBridgeRequest(ctx, req); err != nil {

ctx.Logger().Warn(fmt.Sprintf("error while verifying Bridgetx request, error: %v\n", err))

return ErrNotAuthorized

}

sender := pantherBridge.UnmarshalAddressPB(req.Bridgetx.Sender)

defaultErrMsg := "[BridgeCurrency] failed to process transfer"

pending := &PendingTxs{}

ctx.Get(pendingTXsKey, pending)

for \_, v := range pending.Transactions {

if v.Slot == req.Bridgetx.Slot {

return fmt.Errorf("Error appending Bridge transaction with existing slot -%d", v.Slot)

}

}

pending.Transactions = append(pending.Transactions, req.Bridgetx)

receiver := pantherBridge.UnmarshalAddressPB(req.Bridgetx.NewOwner)

coin, err := loadCoin(ctx, req.Bridgetx.Slot)

if err != nil {

return errors.Wrap(err, defaultErrMsg)

}

ctx.Logger().Debug(fmt.Sprintf("Transfer %v from %v to %v", coin.Slot, sender, receiver))

if err := transferCoin(ctx, coin, sender, receiver); err != nil {

return errors.Wrap(err, defaultErrMsg)

}

return ctx.Set(pendingTXsKey, pending)

}

func (c \*BridgeCurrency) depositRequest(ctx contract.Context, req \*DepositRequest) error {

// : Validate req, must have denomination, from, contract address set

pbk := &BridgeBookKeeping{}

ctx.Get(blockHeightKey, pbk)

pending := &PendingTxs{}

ctx.Get(pendingTXsKey, pending)

// create a new deposit block for the deposit event

tx := &BridgeTx{

Slot: req.Slot,

Denomination: req.Denomination,

NewOwner: req.From,

Proof: make([]byte, 8),

}

pb := &BridgeBlock{

//MerkleHash: merkleHash,

Transactions: []\*BridgeTx{tx},

Uid: req.DepositBlock,

}

// what if the number scheme is not aligned with our internal!!!!

//lets add some tests around this

err := ctx.Set(blockKey(req.DepositBlock.Value), pb)

if err != nil {

return err

}

defaultErrMsg := "[BridgeCurrency] failed to process deposit"

// Update the sender's local Bridge account to reflect the deposit

ownerAddr := pantherBridge.UnmarshalAddressPB(req.From)

ctx.Logger().Debug(fmt.Sprintf("Deposit %v from %v", req.Slot, ownerAddr))

account, err := loadAccount(ctx, ownerAddr)

if err != nil {

return errors.Wrap(err, defaultErrMsg)

}

coin := &Coin{

Slot: req.Slot,

State: CoinState\_DEPOSITED,

Token: req.Denomination,

Contract: req.Contract,

}

err = saveCoin(ctx, coin)

if err != nil {

return errors.Wrap(err, defaultErrMsg)

}

account.Slots = append(account.Slots, req.Slot)

if err = saveAccount(ctx, account); err != nil {

return errors.Wrap(err, defaultErrMsg)

}

c.emitDepositConfirmedEvent(ctx, coin, req.From)

if req.DepositBlock.Value.Cmp(&pbk.CurrentHeight.Value) > 0 {

pbk.CurrentHeight.Value = req.DepositBlock.Value

return ctx.Set(blockHeightKey, pbk)

}

return nil

}

// BalanceOf returns the Bridge coins owned by an entity. The request must specify the address of

// the token contract for which Bridge coins should be returned.

func (c \*BridgeCurrency) BalanceOf(ctx contract.StaticContext, req \*BalanceOfRequest) (\*BalanceOfResponse, error) {

ownerAddr := pantherBridge.UnmarshalAddressPB(req.Owner)

account, err := loadAccount(ctx, ownerAddr)

if err != nil {

return nil, errors.Wrap(err, "[BridgeCurrency] failed to retrieve coin balance")

}

coins := make([]\*Coin, 0, len(account.Slots))

for \_, slot := range account.Slots {

coin, err := loadCoin(ctx, slot)

if err != nil {

ctx.Logger().Error(err.Error())

}

coins = append(coins, coin)

}

return &BalanceOfResponse{Coins: coins}, nil

}

// Reset updates the state of a Bridge coin from EXITING to DEPOSITED

// This method should only be called by the Bridge Currency BridgeAuthority when a coin's exit is successfully challenged

func (c \*BridgeCurrency) coinReset(ctx contract.Context, req \*CoinResetRequest) error {

defaultErrMsg := "[BridgeCurrency] failed to reset coin"

coin, err := loadCoin(ctx, req.Slot)

if err != nil {

return errors.Wrap(err, defaultErrMsg)

}

if coin.State != CoinState\_EXITING {

return fmt.Errorf("[BridgeCurrency] can't reset coin %v in state %s", coin.Slot, coin.State)

}

coin.State = CoinState\_DEPOSITED

if err = saveCoin(ctx, coin); err != nil {

return errors.Wrap(err, defaultErrMsg)

}

c.emitResetConfirmedEvent(ctx, req.Owner, req.Slot)

return nil

}

// ExitCoin updates the state of a Bridge coin from DEPOSITED to EXITING.

// This method should only be called by the Bridge Currency BridgeAuthority when it detects an attempted exit

// of a Bridge coin on Ethereum Mainnet.

func (c \*BridgeCurrency) exitCoin(ctx contract.Context, req \*ExitCoinRequest) error {

defaultErrMsg := "[BridgeCurrency] failed to exit coin"

coin, err := loadCoin(ctx, req.Slot)

if err != nil {

return errors.Wrap(err, defaultErrMsg)

}

if coin.State != CoinState\_DEPOSITED {

return fmt.Errorf("[BridgeCurrency] can't exit coin %v in state %s", coin.Slot, coin.State)

}

coin.State = CoinState\_EXITING

if err = saveCoin(ctx, coin); err != nil {

return errors.Wrap(err, defaultErrMsg)

}

c.emitExitConfirmedEvent(ctx, req.Owner, req.Slot)

return nil

}

// WithdrawCoin removes a Bridge coin from a local Bridge account.

// This method should only be called by the Bridge Currency BridgeAuthority when it detects a withdrawal of a

// Bridge coin on Ethereum Mainnet.

func (c \*BridgeCurrency) withdrawCoin(ctx contract.Context, req \*WithdrawCoinRequest) error {

defaultErrMsg := "[BridgeCurrency] failed to withdraw coin"

coin, err := loadCoin(ctx, req.Slot)

if err != nil {

return errors.Wrap(err, defaultErrMsg)

}

ownerAddr := pantherBridge.UnmarshalAddressPB(req.Owner)

account, err := loadAccount(ctx, ownerAddr)

if err != nil {

return errors.Wrap(err, defaultErrMsg)

}

for i, slot := range account.Slots {

if slot == coin.Slot {

// NOTE: We don't require the coin to be in EXITED state to process the withdrawal

// because the owner is free (in theory) to initiate an exit without involving the

// DAppChain.

account.Slots[i] = account.Slots[len(account.Slots)-1]

account.Slots = account.Slots[:len(account.Slots)-1]

if err = saveAccount(ctx, account); err != nil {

return errors.Wrap(err, defaultErrMsg)

}

ctx.Delete(coinKey(slot))

c.emitWithdrawConfirmedEvent(ctx, coin, req.Owner, req.Slot)

return nil

}

}

return errors.New(defaultErrMsg)

}

func (c \*BridgeCurrency) GetCurrentBlockRequest(ctx contract.StaticContext, req \*GetCurrentBlockRequest) (\*GetCurrentBlockResponse, error) {

pbk := &BridgeBookKeeping{}

ctx.Get(blockHeightKey, pbk)

return &GetCurrentBlockResponse{BlockHeight: pbk.CurrentHeight}, nil

}

func (c \*BridgeCurrency) GetBlockRequest(ctx contract.StaticContext, req \*GetBlockRequest) (\*GetBlockResponse, error) {

pb := &BridgeBlock{}

err := ctx.Get(blockKey(req.BlockHeight.Value), pb)

if err != nil {

return nil, err

}

return &GetBlockResponse{Block: pb}, nil

}

func (c \*BridgeCurrency) GetUserSlotsRequest(ctx contract.StaticContext, req \*GetUserSlotsRequest) (\*GetUserSlotsResponse, error) {

if req.From == nil {

return nil, fmt.Errorf("invalid account parameter")

}

reqAcct, err := loadAccount(ctx, pantherBridge.UnmarshalAddressPB(req.From))

if err != nil {

return nil, err

}

res := &GetUserSlotsResponse{}

res.Slots = reqAcct.Slots

return res, nil

}

func (c \*BridgeCurrency) GetBridgeTxRequest(ctx contract.StaticContext, req \*GetBridgeTxRequest) (\*GetBridgeTxResponse, error) {

pb := &BridgeBlock{}

if req.BlockHeight == nil {

return nil, fmt.Errorf("invalid BlockHeight")

}

err := ctx.Get(blockKey(req.BlockHeight.Value), pb)

if err != nil {

return nil, err

}

leaves := make(map[uint64][]byte)

tx := &BridgeTx{}

for \_, v := range pb.Transactions {

// Merklize tx set

leaves[v.Slot] = v.MerkleHash

// Save the tx matched

if v.Slot == req.Slot {

tx = v

}

}

// Create SMT

smt, err := mamamerkle.NewSparseMerkleTree(64, leaves)

if err != nil {

return nil, err

}

tx.Proof = smt.CreateMerkleProof(req.Slot)

res := &GetBridgeTxResponse{

Bridgetx: tx,

}

return res, nil

}

func (c \*BridgeCurrency) ProcessRequestBatch(ctx contract.Context, req \*pctypes.BridgeCurrencyRequestBatch) error {

if hasPermission, \_ := ctx.HasPermission(SubmitEventsPermission, []string{BridgeAuthorityRole}); !hasPermission {

return ErrNotAuthorized

}

// No requests to process

if len(req.Requests) == 0 {

return nil

}

requestBatchTally := RequestBatchTally{}

if err := ctx.Get(requestBatchTallyKey(), &requestBatchTally); err != nil {

if err != contract.ErrNotFound {

return errors.Wrapf(err, "unable to retrieve event batch tally")

}

}

// We have already consumed all the events being offered.

lastRequest := req.Requests[len(req.Requests)-1]

if isRequestAlreadySeen(lastRequest.Meta, &requestBatchTally) {

return nil

}

var err error

loop:

for \_, request := range req.Requests {

switch data := request.Data.(type) {

case \*pctypes.BridgeCurrencyRequest\_Deposit:

if isRequestAlreadySeen(request.Meta, &requestBatchTally) {

break

}

err = c.depositRequest(ctx, data.Deposit)

if err != nil {

break loop

}

requestBatchTally.LastSeenBlockNumber = request.Meta.BlockNumber

requestBatchTally.LastSeenTxIndex = request.Meta.TxIndex

requestBatchTally.LastSeenLogIndex = request.Meta.LogIndex

case \*pctypes.BridgeCurrencyRequest\_CoinReset:

if isRequestAlreadySeen(request.Meta, &requestBatchTally) {

break

}

err = c.coinReset(ctx, data.CoinReset)

if err != nil {

break loop

}

requestBatchTally.LastSeenBlockNumber = request.Meta.BlockNumber

requestBatchTally.LastSeenTxIndex = request.Meta.TxIndex

requestBatchTally.LastSeenLogIndex = request.Meta.LogIndex

case \*pctypes.BridgeCurrencyRequest\_StartedExit:

if isRequestAlreadySeen(request.Meta, &requestBatchTally) {

break

}

err = c.exitCoin(ctx, data.StartedExit)

if err != nil {

break loop

}

requestBatchTally.LastSeenBlockNumber = request.Meta.BlockNumber

requestBatchTally.LastSeenTxIndex = request.Meta.TxIndex

requestBatchTally.LastSeenLogIndex = request.Meta.LogIndex

case \*pctypes.BridgeCurrencyRequest\_Withdraw:

if isRequestAlreadySeen(request.Meta, &requestBatchTally) {

break

}

err = c.withdrawCoin(ctx, data.Withdraw)

if err != nil {

break loop

}

requestBatchTally.LastSeenBlockNumber = request.Meta.BlockNumber

requestBatchTally.LastSeenTxIndex = request.Meta.TxIndex

requestBatchTally.LastSeenLogIndex = request.Meta.LogIndex

}

}

if err != nil {

return errors.Wrapf(err, "unable to consume one or more requests")

}

if err = ctx.Set(requestBatchTallyKey(), &requestBatchTally); err != nil {

return errors.Wrapf(err, "unable to save request batch tally")

}

return err

}

func loadAccount(ctx contract.StaticContext, owner pantherBridge.Address) (\*Account, error) {

acct := &Account{

Owner: owner.MarshalPB(),

}

err := ctx.Get(accountKey(owner), acct)

if err != nil && err != contract.ErrNotFound {

return nil, err

}

return acct, nil

}

func saveAccount(ctx contract.Context, acct \*Account) error {

owner := pantherBridge.UnmarshalAddressPB(acct.Owner)

return ctx.Set(accountKey(owner), acct)

}

func loadCoin(ctx contract.StaticContext, slot uint64) (\*Coin, error) {

coin := &Coin{}

err := ctx.Get(coinKey(slot), coin)

if err != nil {

return nil, errors.Wrapf(err, "failed to load coin %v", coin.Slot)

}

return coin, nil

}

func saveCoin(ctx contract.Context, coin \*Coin) error {

if err := ctx.Set(coinKey(coin.Slot), coin); err != nil {

return errors.Wrapf(err, "failed to save coin %v", coin.Slot)

}

return nil

}

// Updates the sender's and receiver's local Bridge accounts to reflect a Bridge coin transfer.

func transferCoin(ctx contract.Context, coin \*Coin, sender, receiver pantherBridge.Address) error {

if coin.State != CoinState\_DEPOSITED {

return fmt.Errorf("can't transfer coin %v in state %s", coin.Slot, coin.State)

}

fromAcct, err := loadAccount(ctx, sender)

if err != nil {

return err

}

coinIdx := -1

for i, slot := range fromAcct.Slots {

if slot == coin.Slot {

coinIdx = i

break

}

}

if coinIdx == -1 {

return fmt.Errorf("can't transfer coin %v: sender doesn't own it", coin.Slot)

}

toAcct, err := loadAccount(ctx, receiver)

if err != nil {

return err

}

fromSlots := fromAcct.Slots

toAcct.Slots = append(toAcct.Slots, fromSlots[coinIdx])

fromSlots[coinIdx] = fromSlots[len(fromSlots)-1]

fromAcct.Slots = fromSlots[:len(fromSlots)-1]

if err := saveAccount(ctx, fromAcct); err != nil {

return errors.Wrap(err, "failed to transfer coin %v: can't save sender account")

}

if err := saveAccount(ctx, toAcct); err != nil {

return errors.Wrap(err, "failed to transfer coin %v: can't save receiver account")

}

payload, err := proto.Marshal(&TransferConfirmed{

From: fromAcct.GetOwner(),

To: toAcct.GetOwner(),

Slot: coin.GetSlot(),

})

if err != nil {

return err

}

ctx.EmitTopics(payload, contractBridgeCurrencyTransferConfirmedEventTopic)

return nil

}

func soliditySha3(data uint64) ([]byte, error) {

pairs := []\*evmcompat.Pair{&evmcompat.Pair{"uint64", strconv.FormatUint(data, 10)}}

hash, err := evmcompat.SoliditySHA3(pairs)

if err != nil {

return []byte{}, err

}

return hash, err

}

func rlpEncodeWithSha3(pb \*BridgeTx) ([]byte, error) {

hash, err := rlpEncode(pb)

if err != nil {

return []byte{}, err

}

d := sha3.NewKeccak256()

d.Write(hash)

return d.Sum(nil), nil

}

func isRequestAlreadySeen(meta \*pctypes.BridgeCurrencyEventMeta, currentTally \*RequestBatchTally) bool {

if meta.BlockNumber != currentTally.LastSeenBlockNumber {

return meta.BlockNumber <= currentTally.LastSeenBlockNumber

}

if meta.TxIndex != currentTally.LastSeenTxIndex {

return meta.TxIndex <= currentTally.LastSeenTxIndex

}

if meta.LogIndex != currentTally.LastSeenLogIndex {

return meta.LogIndex <= currentTally.LastSeenLogIndex

}

return true

}

func rlpEncode(pb \*BridgeTx) ([]byte, error) {

return rlp.EncodeToBytes([]interface{}{

uint64(pb.Slot),

pb.PreviousBlock.Value.Bytes(),

uint32(pb.Denomination.Value.Int64()),

pb.GetNewOwner().Local,

})

}

**BridgeAuthority**

**\*\*\*\*\*\*\*\*\*\*\*\*\***

const (

DefaultMaxRetry = 5

DefaultRetryDelay = 1 \* time.Second

)

type sortableRequests struct {

requests []\*pctypes.BridgeCurrencyRequest

}

func (s sortableRequests) Less(i, j int) bool {

if s.requests[i].Meta.BlockNumber != s.requests[j].Meta.BlockNumber {

return s.requests[i].Meta.BlockNumber < s.requests[j].Meta.BlockNumber

}

if s.requests[i].Meta.TxIndex != s.requests[j].Meta.TxIndex {

return s.requests[i].Meta.TxIndex < s.requests[j].Meta.TxIndex

}

if s.requests[i].Meta.LogIndex != s.requests[j].Meta.LogIndex {

return s.requests[i].Meta.LogIndex < s.requests[j].Meta.LogIndex

}

return i < j

}

func (s sortableRequests) Len() int {

return len(s.requests)

}

func (s sortableRequests) Swap(i, j int) {

tmpRequest := s.requests[i]

s.requests[i] = s.requests[j]

s.requests[j] = tmpRequest

}

func (s sortableRequests) PrepareRequestBatch() \*pctypes.BridgeCurrencyRequestBatch {

requestBatch := &pctypes.BridgeCurrencyRequestBatch{}

sort.Sort(s)

requestBatch.Requests = s.requests

return requestBatch

}

type BridgeAuthorityConfig struct {

// Each Bridge block number must be a multiple of this value

BridgeBlockInterval uint32

StatusServiceAddress string

DAppChainClientCfg DAppChainBridgeClientConfig

EthClientCfg eth.EthBridgeClientConfig

}

type BridgeBlockWorkerStatus struct {

LastSeenDAppChainBridgeBlockNum \*big.Int

LastSeenEthBridgeBlockNum \*big.Int

// Just to avoid hassle of looking into yaml file

BridgeBlockInterval uint32

}

// BridgeBlockWorker sends non-deposit Bridge block from the DAppChain to Ethereum.

type BridgeBlockWorker struct {

ethBridgeClient eth.EthBridgeClient

dappBridgeClient DAppChainBridgeClient

BridgeBlockInterval uint32

statusRwMutex sync.RWMutex

status BridgeBlockWorkerStatus

}

func NewBridgeBlockWorker(cfg \*BridgeAuthorityConfig) \*BridgeBlockWorker {

return &BridgeBlockWorker{

ethBridgeClient: &eth.EthBridgeClientImpl{EthBridgeClientConfig: cfg.EthClientCfg},

dappBridgeClient: &DAppChainBridgeClientImpl{DAppChainBridgeClientConfig: cfg.DAppChainClientCfg},

BridgeBlockInterval: cfg.BridgeBlockInterval,

status: BridgeBlockWorkerStatus{

BridgeBlockInterval: cfg.BridgeBlockInterval,

},

}

}

func (w \*BridgeBlockWorker) Init() error {

if err := w.ethBridgeClient.Init(); err != nil {

return err

}

return w.dappBridgeClient.Init()

}

func (w \*BridgeBlockWorker) Status() BridgeBlockWorkerStatus {

w.statusRwMutex.RLock()

defer w.statusRwMutex.RUnlock()

return w.status

}

func (w \*BridgeBlockWorker) Run() {

go runWithRecovery(func() {

loopWithInterval(w.sendBridgeBlocksToEthereum, 5\*time.Second)

})

}

// DAppChain -> Bridge Blocks -> Ethereum

func (w \*BridgeBlockWorker) sendBridgeBlocksToEthereum() error {

pendingTxs, err := w.dappBridgeClient.GetPendingTxs()

if err != nil {

return errors.Wrap(err, "failed to get pending transactions")

}

// Only call SubmitBlockToMainnet, if pending transactions are there.

if len(pendingTxs.Transactions) > 0 {

if err = w.dappBridgeClient.FinalizeCurrentBridgeBlock(); err != nil {

return errors.Wrap(err, "failed to finalize current Bridge block")

}

}

if err = w.syncBridgeBlocksWithEthereum(); err != nil {

return errors.Wrap(err, "failed to sync Bridge blocks with mainnet")

}

return nil

}

// Send any finalized but unsubmitted Bridge blocks from the DAppChain to Ethereum.

func (w \*BridgeBlockWorker) syncBridgeBlocksWithEthereum() error {

curEthBridgeBlockNum, err := w.ethBridgeClient.CurrentBridgeBlockNum()

if err != nil {

return err

}

log.Printf("solBridge.CurrentBlock: %s", curEthBridgeBlockNum.String())

curPantherBridgeBridgeBlockNum, err := w.dappBridgeClient.CurrentBridgeBlockNum()

if err != nil {

return err

}

w.statusRwMutex.Lock()

w.status.LastSeenEthBridgeBlockNum = curEthBridgeBlockNum

w.status.LastSeenDAppChainBridgeBlockNum = curPantherBridgeBridgeBlockNum

w.statusRwMutex.Unlock()

if curPantherBridgeBridgeBlockNum.Cmp(curEthBridgeBlockNum) == 0 {

// DAppChain and Ethereum both have all the finalized Bridge blocks

return nil

}

BridgeBlockInterval := big.NewInt(int64(w.BridgeBlockInterval))

unsubmittedBridgeBlockNum := nextBridgeBlockNum(curEthBridgeBlockNum, BridgeBlockInterval)

if unsubmittedBridgeBlockNum.Cmp(curPantherBridgeBridgeBlockNum) > 0 {

// All the finalized Bridge blocks in the DAppChain have been submitted to Ethereum

return nil

}

block, err := w.dappBridgeClient.BridgeBlockAt(unsubmittedBridgeBlockNum)

if err != nil {

return err

}

if err := w.submitBridgeBlockToEthereum(unsubmittedBridgeBlockNum, block.MerkleHash); err != nil {

return err

}

return nil

}

// Submits a Bridge block (or rather its merkle root) to the Bridge Solidity contract on Ethereum.

// This function will block until the tx is confirmed, or times out.

func (w \*BridgeBlockWorker) submitBridgeBlockToEthereum(BridgeBlockNum \*big.Int, merkleRoot []byte) error {

curEthBridgeBlockNum, err := w.ethBridgeClient.CurrentBridgeBlockNum()

if err != nil {

return err

}

// Try to avoid submitting the same Bridge blocks multiple times

if BridgeBlockNum.Cmp(curEthBridgeBlockNum) <= 0 {

return nil

}

if len(merkleRoot) != 32 {

return errors.New("invalid merkle root size")

}

var root [32]byte

copy(root[:], merkleRoot)

log.Printf("\*\*\*\*\*\*\*\*\* #### Submitting BridgeBlockNum: %s with root: %v", BridgeBlockNum.String(), root)

return w.ethBridgeClient.SubmitBridgeBlock(BridgeBlockNum, root)

}

type BridgeCoinWorkerStatus struct {

DepositEventsProcessed int

WithdrawEventsProcessed int

StartedExitEventsProcessed int

CoinResetEventsProcessed int

LastSeenEthBlockNumber uint64

LastReportedRequestBatchTally \*pctypes.BridgeCurrencyRequestBatchTally

}

// BridgeCoinWorker sends Bridge deposits from Ethereum to the DAppChain.

type BridgeCoinWorker struct {

ethBridgeClient eth.EthBridgeClient

dappBridgeClient DAppChainBridgeClient

statusRwMutex sync.RWMutex

status BridgeCoinWorkerStatus

}

func NewBridgeCoinWorker(cfg \*BridgeAuthorityConfig) \*BridgeCoinWorker {

return &BridgeCoinWorker{

ethBridgeClient: &eth.EthBridgeClientImpl{EthBridgeClientConfig: cfg.EthClientCfg},

dappBridgeClient: &DAppChainBridgeClientImpl{DAppChainBridgeClientConfig: cfg.DAppChainClientCfg},

status: BridgeCoinWorkerStatus{},

}

}

func (w \*BridgeCoinWorker) Init() error {

if err := w.ethBridgeClient.Init(); err != nil {

return err

}

return w.dappBridgeClient.Init()

}

func (w \*BridgeCoinWorker) Run() {

go runWithRecovery(func() {

loopWithInterval(w.sendCoinEventsToDAppChain, 4\*time.Second)

})

}

func (w \*BridgeCoinWorker) Status() BridgeCoinWorkerStatus {

w.statusRwMutex.RLock()

defer w.statusRwMutex.RUnlock()

return w.status

}

func (w \*BridgeCoinWorker) sendCoinEventsToDAppChain() error {

tally, err := w.dappBridgeClient.GetRequestBatchTally()

if err != nil {

return errors.Wrapf(err, "failed to fetch current request batch tally from dappchain")

}

// If HasSeenAnyRequest is false means we havent seen any

// block, so set startEthBlock to zero only, otherwise

// set it to lastSeen + 1

var startEthBlock uint64 = 0

if tally.LastSeenBlockNumber != 0 {

startEthBlock = tally.LastSeenBlockNumber + 1

}

// : limit max block range per batch

latestEthBlock, err := w.ethBridgeClient.LatestEthBlockNum()

if err != nil {

return errors.Wrapf(err, "failed to fetch latest block number for eth contract")

}

w.statusRwMutex.Lock()

w.status.LastSeenEthBlockNumber = latestEthBlock

w.status.LastReportedRequestBatchTally = tally

w.statusRwMutex.Unlock()

if latestEthBlock < startEthBlock {

// Wait for Ethereum to produce a new block...

return nil

}

// We need to retrieve all events first, and then apply them in correct order

// to make sure, we apply events in proper order to dappchain

depositEvents, err := w.ethBridgeClient.FetchDeposits(startEthBlock, latestEthBlock)

if err != nil {

return errors.Wrap(err, "failed to fetch Bridge deposit events from Ethereum")

}

withdrewEvents, err := w.ethBridgeClient.FetchWithdrews(startEthBlock, latestEthBlock)

if err != nil {

return errors.Wrap(err, "failed to fetch Bridge withdrew events from Ethereum")

}

startedExitEvents, err := w.ethBridgeClient.FetchStartedExit(startEthBlock, latestEthBlock)

if err != nil {

return errors.Wrap(err, "failed to fetch Bridge started exit event from Ethereum")

}

coinResetEvents, err := w.ethBridgeClient.FetchCoinReset(startEthBlock, latestEthBlock)

if err != nil {

return errors.Wrap(err, "failed to fetch Bridge coin reset event from Ethereum")

}

requests := make([]\*pctypes.BridgeCurrencyRequest, len(depositEvents)+len(withdrewEvents)+len(startedExitEvents)+len(coinResetEvents))

i := 0

for \_, event := range depositEvents {

requests[i] = &pctypes.BridgeCurrencyRequest{

Data: &pctypes.BridgeCurrencyRequest\_Deposit{&pctypes.DepositRequest{

Slot: event.Slot,

DepositBlock: event.DepositBlock,

Denomination: event.Denomination,

From: event.From,

Contract: event.Contract,

}},

Meta: event.Meta,

}

i++

}

for \_, event := range withdrewEvents {

requests[i] = &pctypes.BridgeCurrencyRequest{

Data: &pctypes.BridgeCurrencyRequest\_Withdraw{&pctypes.BridgeCurrencyWithdrawCoinRequest{

Owner: event.Owner,

Slot: event.Slot,

}},

Meta: event.Meta,

}

i++

}

for \_, event := range startedExitEvents {

requests[i] = &pctypes.BridgeCurrencyRequest{

Data: &pctypes.BridgeCurrencyRequest\_StartedExit{&pctypes.BridgeCurrencyExitCoinRequest{

Owner: event.Owner,

Slot: event.Slot,

}},

Meta: event.Meta,

}

i++

}

for \_, event := range coinResetEvents {

requests[i] = &pctypes.BridgeCurrencyRequest{

Data: &pctypes.BridgeCurrencyRequest\_CoinReset{&pctypes.BridgeCurrencyCoinResetRequest{

Owner: event.Owner,

Slot: event.Slot,

}},

Meta: event.Meta,

}

i++

}

// No requests to process

if len(requests) == 0 {

return nil

}

requestBatch := sortableRequests{requests: requests}.PrepareRequestBatch()

err = w.dappBridgeClient.ProcessRequestBatch(requestBatch)

if err != nil {

return errors.Wrapf(err, "unable to send request batch to dappchain")

}

w.statusRwMutex.Lock()

w.status.DepositEventsProcessed += len(depositEvents)

w.status.WithdrawEventsProcessed += len(withdrewEvents)

w.status.StartedExitEventsProcessed += len(startedExitEvents)

w.status.CoinResetEventsProcessed += len(coinResetEvents)

w.statusRwMutex.Unlock()

return nil

}

type BridgeAuthorityStatus struct {

CoinWorkerStatus BridgeCoinWorkerStatus

BlockWorkerStatus BridgeBlockWorkerStatus

}

type BridgeAuthority struct {

cfg \*BridgeAuthorityConfig

coinWorker \*BridgeCoinWorker

blockWorker \*BridgeBlockWorker

}

func NewBridgeAuthority(cfg \*BridgeAuthorityConfig) \*BridgeAuthority {

return &BridgeAuthority{

cfg: cfg,

coinWorker: NewBridgeCoinWorker(cfg),

blockWorker: NewBridgeBlockWorker(cfg),

}

}

func (orc \*BridgeAuthority) Status() \*BridgeAuthorityStatus {

return &BridgeAuthorityStatus{

CoinWorkerStatus: orc.coinWorker.Status(),

BlockWorkerStatus: orc.blockWorker.Status(),

}

}

func (orc \*BridgeAuthority) Init() error {

if err := orc.coinWorker.Init(); err != nil {

return err

}

return orc.blockWorker.Init()

}

// : Graceful shutdown

func (orc \*BridgeAuthority) Run() {

go runWithRecovery(func() {

counter := 0

loopWithInterval(func() error {

counter += 1

if counter == 6 { // Submit blocks 6 times less often than fetching events (12 sec)

err := orc.blockWorker.sendBridgeBlocksToEthereum()

if err != nil {

log.Printf("[PCBridgeAuthority] error while sending Bridge blocks to ethereum: %v\n", err)

}

counter = 0

}

err := orc.coinWorker.sendCoinEventsToDAppChain()

if err != nil {

log.Printf("[PCBridgeAuthority] error while sending coin events to dappchain: %v\n", err)

}

return err

}, 2\*time.Second)

})

}

// runWithRecovery should run in a goroutine, it will ensure the given function keeps on running in

// a goroutine as long as it doesn't panic due to a runtime error.

func runWithRecovery(run func()) {

defer func() {

if r := recover(); r != nil {

log.Printf("recovered from panic in a Bridge BridgeAuthority worker: %v\n", r)

// Unless it's a runtime error restart the goroutine

if \_, ok := r.(runtime.Error); !ok {

time.Sleep(30 \* time.Second)

log.Printf("Restarting Bridge BridgeAuthority worker...")

go runWithRecovery(run)

}

}

}()

run()

}

// loopWithInterval will execute the step function in an endless loop while ensuring that each

// loop iteration takes up the minimum specified duration.

func loopWithInterval(step func() error, minStepDuration time.Duration) {

for {

start := time.Now()

if err := step(); err != nil {

log.Println(err)

}

diff := time.Now().Sub(start)

if diff < minStepDuration {

time.Sleep(minStepDuration - diff)

}

}

}

// Computes the block number of the next non-deposit Bridge block.

// The current Bridge block number can be for a deposit or non-deposit Bridge block.

// Bridge block numbers of non-deposit blocks are expected to be multiples of the specified interval.

func nextBridgeBlockNum(current \*big.Int, interval \*big.Int) \*big.Int {

r := current

r.Div(r, interval)

r.Add(r, big.NewInt(1))

return r.Mul(r, interval)

}

**Bridge block synchronizer unit test**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

var (

addr1 = pantherBridge.MustParseAddress("chain:0xb16a379ec18d4093666f8f38b11a3071c920207d")

addr2 = pantherBridge.MustParseAddress("chain:0xfa4c7920accfd66b86f5fd0e69682a79f762d49e")

addr3 = pantherBridge.MustParseAddress("chain:0x5cecd1f7261e1f4c684e297be3edf03b825e01c4")

)

// add test to verify idempodency

func TestRound(t \*testing.T) {

// change to bigint

res := round(9, int64(1000))

assert.Equal(t, res, int64(1000))

res = round(999, 1000)

assert.Equal(t, res, int64(1000))

res = round(0, 1000)

assert.Equal(t, res, int64(1000))

res = round(1000, 1000)

assert.Equal(t, res, int64(2000))

res = round(1001, 1000)

assert.Equal(t, res, int64(2000))

res = round(1999, 1000)

assert.Equal(t, res, int64(2000))

res = round(2000, 1000)

assert.Equal(t, res, int64(3000))

res = round(2001, 1000)

assert.Equal(t, res, int64(3000))

}

func TestBridgeCurrencySMT(t \*testing.T) {

Ctx := plugin.CreateContext(addr1, addr1)

addressMapperAddress := Ctx.CreateContract(address\_mapper.Contract)

Ctx.RegisterContract("addressmapper", addressMapperAddress, addressMapperAddress)

ctx := contractpb.WrapPluginContext(

Ctx,

)

contract := &BridgeCurrency{}

err := contract.Init(ctx, &InitRequest{

BridgeAuthority: addr1.MarshalPB(),

})

require.Nil(t, err)

pending := &PendingTxs{}

ctx.Get(pendingTXsKey, pending)

assert.Equal(t, len(pending.Transactions), 0, "length should be zero")

contractAddr := pantherBridge.RootAddress("eth")

require.Nil(t, saveCoin(ctx, &Coin{

Slot: 5,

Contract: contractAddr.MarshalPB(),

}))

req := &BridgeTxRequest{

Bridgetx: &BridgeTx{

Slot: 5,

NewOwner: addr3.MarshalPB(),

PreviousBlock: &types.BigUInt{Value: \*pantherBridge.NewBigUIntFromInt(0)},

Denomination: &types.BigUInt{Value: \*pantherBridge.NewBigUIntFromInt(0)},

},

}

generatedSender, err := setupBridgeTxAuth(ctx, req.Bridgetx, addr1)

require.Nil(t, err)

err = saveAccount(ctx, &Account{

Owner: generatedSender.MarshalPB(),

Slots: []uint64{5},

})

require.Nil(t, err)

err = contract.BridgeTxRequest(ctx, req)

require.Nil(t, err)

ctx.Get(pendingTXsKey, pending)

assert.Equal(t, len(pending.Transactions), 1, "length should be one")

reqMainnet := &SubmitBlockToMainnetRequest{}

\_, err = contract.SubmitBlockToMainnet(ctx, reqMainnet)

require.Nil(t, err)

require.NotNil(t, Ctx.Events[0])

assert.Equal(t, Ctx.Events[0].Topics[0], "event:BridgeCurrencyTransferConfirmed", "incorrect topic")

transferConfirmed := TransferConfirmed{}

err = proto.Unmarshal(Ctx.Events[0].Event, &transferConfirmed)

require.Nil(t, err)

assert.Equal(t, pantherBridge.UnmarshalAddressPB(transferConfirmed.From).String(), generatedSender.String())

assert.Equal(t, pantherBridge.UnmarshalAddressPB(transferConfirmed.To).String(), addr3.String())

// assert.Equal(t, Ctx.Events[0].Event, []byte("asdfb"), "incorrect merkle hash")

//Ok lets get the same block back

reqBlock := &GetBlockRequest{

BlockHeight: &types.BigUInt{

Value: \*pantherBridge.NewBigUIntFromInt(1000),

},

}

resblock, err := contract.GetBlockRequest(ctx, reqBlock)

require.Nil(t, err)

require.NotNil(t, resblock)

assert.Equal(t, 1, len(resblock.Block.Transactions), "incorrect number of saved transactions")

reqMainnet2 := &SubmitBlockToMainnetRequest{}

\_, err = contract.SubmitBlockToMainnet(ctx, reqMainnet2)

require.Nil(t, err)

reqBlock2 := &GetBlockRequest{}

reqBlock2.BlockHeight = &types.BigUInt{

Value: \*pantherBridge.NewBigUIntFromInt(1000),

}

resblock2, err := contract.GetBlockRequest(ctx, reqBlock2)

require.Nil(t, err)

require.NotNil(t, resblock2)

}

func TestEmptyBridgeBlock(t \*testing.T) {

Ctx := plugin.CreateContext(addr1, addr1)

ctx := contractpb.WrapPluginContext(

Ctx,

)

contract := &BridgeCurrency{}

err := contract.Init(ctx, &InitRequest{

BridgeAuthority: addr1.MarshalPB(),

})

require.Nil(t, err)

pbk := &BridgeBookKeeping{}

ctx.Get(blockHeightKey, pbk)

assert.Equal(t, pbk.CurrentHeight.Value.Int64(), int64(0), "invalid height")

reqMainnet := &SubmitBlockToMainnetRequest{}

\_, err = contract.SubmitBlockToMainnet(ctx, reqMainnet)

require.Nil(t, err)

ctx.Get(blockHeightKey, pbk)

assert.Equal(t, int64(0), pbk.CurrentHeight.Value.Int64(), "invalid height")

reqMainnet = &SubmitBlockToMainnetRequest{}

\_, err = contract.SubmitBlockToMainnet(ctx, reqMainnet)

require.Nil(t, err)

ctx.Get(blockHeightKey, pbk)

assert.Equal(t, int64(0), pbk.CurrentHeight.Value.Int64(), "invalid height")

reqMainnet = &SubmitBlockToMainnetRequest{}

\_, err = contract.SubmitBlockToMainnet(ctx, reqMainnet)

require.Nil(t, err)

ctx.Get(blockHeightKey, pbk)

assert.Equal(t, int64(0), pbk.CurrentHeight.Value.Int64(), "invalid height")

}

func TestSha3Encodings(t \*testing.T) {

res, err := soliditySha3(5)

assert.Equal(t, fmt.Sprintf("0x%x", res), "0xfe07a98784cd1850eae35ede546d7028e6bf9569108995fc410868db775e5e6a", "incorrect sha3 hex")

require.Nil(t, err)

res2, err := soliditySha3(3)

assert.Equal(t, fmt.Sprintf("0x%x", res2), "0xd4c69e49e83a6047f46e42b2d053a1f0c6e70ea42862e5ef4ad66b3666c5e2af", "incorrect sha3 hex")

require.Nil(t, err)

}

func TestRLPEncodings(t \*testing.T) {

address, err := pantherBridge.LocalAddressFromHexString("0x5194b63f10691e46635b27925100cfc0a5ceca62")

require.Nil(t, err)

BridgeTx := &BridgeTx{

Slot: 5,

PreviousBlock: &types.BigUInt{

Value: \*pantherBridge.NewBigUIntFromInt(0),

},

Denomination: &types.BigUInt{

Value: \*pantherBridge.NewBigUIntFromInt(1),

},

NewOwner: &types.Address{ChainId: "default",

Local: address},

}

data, err := rlpEncode(BridgeTx)

assert.Equal(t, "d8058001945194b63f10691e46635b27925100cfc0a5ceca62", fmt.Sprintf("%x", data), "incorrect sha3 hex")

require.Nil(t, err)

}

// Clear pending txs from state after finalizing a block in SubmitBlockToMainnet.

func TestBridgeClearPending(t \*testing.T) {

Ctx := plugin.CreateContext(addr1, addr1)

addressMapperAddress := Ctx.CreateContract(address\_mapper.Contract)

Ctx.RegisterContract("addressmapper", addressMapperAddress, addressMapperAddress)

ctx := contractpb.WrapPluginContext(

Ctx,

)

contract := &BridgeCurrency{}

err := contract.Init(ctx, &InitRequest{

BridgeAuthority: addr1.MarshalPB(),

})

require.Nil(t, err)

pending := &PendingTxs{}

ctx.Get(pendingTXsKey, pending)

assert.Equal(t, len(pending.Transactions), 0, "length should be zero")

contractAddr := pantherBridge.RootAddress("eth")

require.Nil(t, saveCoin(ctx, &Coin{

Slot: 5,

Contract: contractAddr.MarshalPB(),

}))

req := &BridgeTxRequest{

Bridgetx: &BridgeTx{

Slot: 5,

NewOwner: addr3.MarshalPB(),

PreviousBlock: &types.BigUInt{Value: \*pantherBridge.NewBigUIntFromInt(0)},

Denomination: &types.BigUInt{Value: \*pantherBridge.NewBigUIntFromInt(0)},

},

}

generatedSender, err := setupBridgeTxAuth(ctx, req.Bridgetx, addr1)

require.Nil(t, err)

err = saveAccount(ctx, &Account{

Owner: generatedSender.MarshalPB(),

Slots: []uint64{5},

})

require.Nil(t, err)

err = contract.BridgeTxRequest(ctx, req)

require.Nil(t, err)

ctx.Get(pendingTXsKey, pending)

assert.Equal(t, len(pending.Transactions), 1, "length should be one")

reqMainnet := &SubmitBlockToMainnetRequest{}

\_, err = contract.SubmitBlockToMainnet(ctx, reqMainnet)

require.Nil(t, err)

pending2 := &PendingTxs{}

ctx.Get(pendingTXsKey, pending2)

assert.Equal(t, len(pending2.Transactions), 0, "length should be zero")

}

// Error out if an attempt is made to add a tx with a slot that is already referenced in pending txs in BridgeTxRequest.

func TestBridgeErrorDuplicate(t \*testing.T) {

Ctx := plugin.CreateContext(addr1, addr1)

addressMapperAddress := Ctx.CreateContract(address\_mapper.Contract)

Ctx.RegisterContract("addressmapper", addressMapperAddress, addressMapperAddress)

ctx := contractpb.WrapPluginContext(

Ctx,

)

contract := &BridgeCurrency{}

err := contract.Init(ctx, &InitRequest{

BridgeAuthority: addr1.MarshalPB(),

})

require.Nil(t, err)

pending := &PendingTxs{}

ctx.Get(pendingTXsKey, pending)

assert.Equal(t, len(pending.Transactions), 0, "length should be zero")

contractAddr := pantherBridge.RootAddress("eth")

require.Nil(t, saveCoin(ctx, &Coin{

Slot: 5,

Contract: contractAddr.MarshalPB(),

}))

req := &BridgeTxRequest{

Bridgetx: &BridgeTx{

Slot: 5,

NewOwner: addr3.MarshalPB(),

PreviousBlock: &types.BigUInt{Value: \*pantherBridge.NewBigUIntFromInt(0)},

Denomination: &types.BigUInt{Value: \*pantherBridge.NewBigUIntFromInt(0)},

},

}

generatedSender, err := setupBridgeTxAuth(ctx, req.Bridgetx, addr1)

require.Nil(t, err)

err = saveAccount(ctx, &Account{

Owner: generatedSender.MarshalPB(),

Slots: []uint64{5},

})

require.Nil(t, err)

err = contract.BridgeTxRequest(ctx, req)

require.Nil(t, err)

//Send slot5 a second time

err = contract.BridgeTxRequest(ctx, req)

require.NotNil(t, err)

}

func TestBridgeCurrencyBalanceAfterDeposit(t \*testing.T) {

BridgeContract, ctx := getBridgeContractAndContext(t)

tokenIDs := []\*types.BigUInt{

&types.BigUInt{Value: \*pantherBridge.NewBigUIntFromInt(721)},

&types.BigUInt{Value: \*pantherBridge.NewBigUIntFromInt(127)},

}

err := BridgeContract.ProcessRequestBatch(ctx, &pctypes.BridgeCurrencyRequestBatch{

Requests: []\*pctypes.BridgeCurrencyRequest{

&pctypes.BridgeCurrencyRequest{

Data: &pctypes.BridgeCurrencyRequest\_Deposit{&pctypes.DepositRequest{

Slot: 123,

DepositBlock: &types.BigUInt{Value: \*pantherBridge.NewBigUIntFromInt(3)},

Denomination: tokenIDs[0],

From: addr2.MarshalPB(),

Contract: addr3.MarshalPB(),

}},

Meta: &pctypes.BridgeCurrencyEventMeta{

BlockNumber: 1,

LogIndex: 0,

TxIndex: 0,

},

},

},

})

require.Nil(t, err)

resp, err := BridgeContract.BalanceOf(ctx, &BalanceOfRequest{

Owner: addr2.MarshalPB(),

Contract: addr3.MarshalPB(),

})

require.Nil(t, err)

require.Len(t, resp.Coins, 1, "account should have one coin")

correctCoin := &Coin{

Slot: 123,

State: CoinState\_DEPOSITED,

Token: tokenIDs[0],

Contract: addr3.MarshalPB(),

}

assert.Equal(t, resp.Coins[0].String(), correctCoin.String())

err = BridgeContract.ProcessRequestBatch(ctx, &pctypes.BridgeCurrencyRequestBatch{

Requests: []\*pctypes.BridgeCurrencyRequest{

&pctypes.BridgeCurrencyRequest{

Data: &pctypes.BridgeCurrencyRequest\_Deposit{&pctypes.DepositRequest{

Slot: 456,

DepositBlock: &types.BigUInt{Value: \*pantherBridge.NewBigUIntFromInt(3)},

Denomination: tokenIDs[1],

From: addr2.MarshalPB(),

Contract: addr3.MarshalPB(),

}},

Meta: &pctypes.BridgeCurrencyEventMeta{

BlockNumber: 2,

LogIndex: 0,

TxIndex: 0,

},

},

},

})

require.Nil(t, err)

resp, err = BridgeContract.BalanceOf(ctx, &BalanceOfRequest{

Owner: addr2.MarshalPB(),

Contract: addr3.MarshalPB(),

})

require.Nil(t, err)

correntCoin1 := &Coin{

Slot: 123,

State: CoinState\_DEPOSITED,

Token: tokenIDs[0],

Contract: addr3.MarshalPB(),

}

correntCoin2 := &Coin{

Slot: 456,

State: CoinState\_DEPOSITED,

Token: tokenIDs[1],

Contract: addr3.MarshalPB(),

}

assert.Equal(t, 2, len(resp.Coins))

assert.Equal(t, correntCoin1.String(), resp.Coins[0].String())

assert.Equal(t, correntCoin2.String(), resp.Coins[1].String())

}

func TestBridgeCurrencyTransferWithInvalidSender(t \*testing.T) {

BridgeContract, ctx := getBridgeContractAndContext(t)

req := &BridgeTxRequest{

Bridgetx: &BridgeTx{

Slot: 5, // sender doesn't own this coin

NewOwner: addr3.MarshalPB(),

PreviousBlock: &types.BigUInt{Value: \*pantherBridge.NewBigUIntFromInt(0)},

Denomination: &types.BigUInt{Value: \*pantherBridge.NewBigUIntFromInt(0)},

},

}

\_, err := setupBridgeTxAuth(ctx, req.Bridgetx, addr1)

require.Nil(t, err)

err = BridgeContract.BridgeTxRequest(ctx, req)

require.NotNil(t, err)

}

func TestBridgeCurrencyTxAuth(t \*testing.T) {

BridgeContract, ctx := getBridgeContractAndContext(t)

contractAddr := pantherBridge.RootAddress("eth")

require.Nil(t, saveCoin(ctx, &Coin{

Slot: 5,

Contract: contractAddr.MarshalPB(),

}))

req := &BridgeTxRequest{

Bridgetx: &BridgeTx{

Slot: 5,

NewOwner: addr3.MarshalPB(),

PreviousBlock: &types.BigUInt{Value: \*pantherBridge.NewBigUIntFromInt(0)},

Denomination: &types.BigUInt{Value: \*pantherBridge.NewBigUIntFromInt(0)},

},

}

// Map addr2 against ethAddress instead of addr1

generatedSender, err := setupBridgeTxAuth(ctx, req.Bridgetx, addr2)

require.Nil(t, err)

err = saveAccount(ctx, &Account{

Owner: generatedSender.MarshalPB(),

Slots: []uint64{5},

})

require.Nil(t, err)

// request wont go through as mapping wont be found

err = BridgeContract.BridgeTxRequest(ctx, req)

require.Equal(t, err, ErrNotAuthorized)

}

func TestBridgeCurrencyTransferWithInvalidCoinState(t \*testing.T) {

BridgeContract, ctx := getBridgeContractAndContext(t)

ethPrivKey, err := crypto.GenerateKey()

require.Nil(t, err)

coins := []\*Coin{

&Coin{Slot: 5, State: CoinState\_EXITING},

&Coin{Slot: 6, State: CoinState\_CHALLENGED},

&Coin{Slot: 7, State: CoinState\_EXITED},

}

for \_, coin := range coins {

require.Nil(t, saveCoin(ctx, coin))

}

err = saveAccount(ctx, &Account{

Owner: addr2.MarshalPB(),

Slots: []uint64{5, 6, 7},

})

require.Nil(t, err)

for \_, coin := range coins {

req := &BridgeTxRequest{

Bridgetx: &BridgeTx{

Slot: coin.Slot,

NewOwner: addr3.MarshalPB(),

PreviousBlock: &types.BigUInt{Value: \*pantherBridge.NewBigUIntFromInt(0)},

Denomination: &types.BigUInt{Value: \*pantherBridge.NewBigUIntFromInt(0)},

},

}

\_, err := setupBridgeTxAuthWithKey(ctx, ethPrivKey, req.Bridgetx, addr1)

require.Nil(t, err)

err = BridgeContract.BridgeTxRequest(ctx, req)

require.NotNil(t, err)

}

}

func TestBridgeCurrencyExitWithInvalidCoinState(t \*testing.T) {

BridgeContract, ctx := getBridgeContractAndContext(t)

coins := []\*Coin{

&Coin{Slot: 5, State: CoinState\_EXITING},

&Coin{Slot: 6, State: CoinState\_CHALLENGED},

&Coin{Slot: 7, State: CoinState\_EXITED},

}

for \_, coin := range coins {

require.Nil(t, saveCoin(ctx, coin))

}

err := saveAccount(ctx, &Account{

Owner: addr2.MarshalPB(),

Slots: []uint64{5, 6, 7},

})

require.Nil(t, err)

for i, coin := range coins {

err = BridgeContract.ProcessRequestBatch(ctx, &pctypes.BridgeCurrencyRequestBatch{

Requests: []\*pctypes.BridgeCurrencyRequest{

&pctypes.BridgeCurrencyRequest{

Data: &pctypes.BridgeCurrencyRequest\_StartedExit{&pctypes.BridgeCurrencyExitCoinRequest{

Owner: addr2.MarshalPB(),

Slot: coin.Slot,

}},

Meta: &pctypes.BridgeCurrencyEventMeta{

BlockNumber: uint64(i) + 1,

LogIndex: 0,

TxIndex: 0,

},

},

},

})

require.NotNil(t, err)

}

}

func TestBridgeCurrencyExit(t \*testing.T) {

BridgeContract, ctx := getBridgeContractAndContext(t)

contractAddr := pantherBridge.RootAddress("eth")

coins := []\*Coin{

&Coin{Slot: 5, State: CoinState\_DEPOSITED, Contract: contractAddr.MarshalPB()},

&Coin{Slot: 6, State: CoinState\_DEPOSITED, Contract: contractAddr.MarshalPB()},

&Coin{Slot: 7, State: CoinState\_DEPOSITED, Contract: contractAddr.MarshalPB()},

}

for \_, coin := range coins {

require.Nil(t, saveCoin(ctx, coin))

}

err := saveAccount(ctx, &Account{

Owner: addr2.MarshalPB(),

Slots: []uint64{5, 6, 7},

})

require.Nil(t, err)

err = BridgeContract.ProcessRequestBatch(ctx, &pctypes.BridgeCurrencyRequestBatch{

Requests: []\*pctypes.BridgeCurrencyRequest{

&pctypes.BridgeCurrencyRequest{

Data: &pctypes.BridgeCurrencyRequest\_StartedExit{&pctypes.BridgeCurrencyExitCoinRequest{

Owner: addr2.MarshalPB(),

Slot: coins[1].Slot,

}},

Meta: &pctypes.BridgeCurrencyEventMeta{

BlockNumber: 1,

LogIndex: 0,

TxIndex: 0,

},

},

},

})

require.Nil(t, err)

resp, err := BridgeContract.BalanceOf(ctx, &BalanceOfRequest{

Owner: addr2.MarshalPB(),

Contract: contractAddr.MarshalPB(),

})

require.Nil(t, err)

assert.Equal(t, &Coin{

Slot: 6,

State: CoinState\_EXITING,

Contract: contractAddr.MarshalPB(),

}, resp.Coins[1])

}

func TestBridgeCurrencyWithdraw(t \*testing.T) {

BridgeContract, ctx := getBridgeContractAndContext(t)

contractAddr := pantherBridge.RootAddress("eth")

coins := []\*Coin{

&Coin{Slot: 5, State: CoinState\_DEPOSITED, Contract: contractAddr.MarshalPB()},

&Coin{Slot: 6, State: CoinState\_EXITING, Contract: contractAddr.MarshalPB()},

&Coin{Slot: 7, State: CoinState\_CHALLENGED, Contract: contractAddr.MarshalPB()},

&Coin{Slot: 8, State: CoinState\_EXITED, Contract: contractAddr.MarshalPB()},

}

for \_, coin := range coins {

require.Nil(t, saveCoin(ctx, coin))

}

err := saveAccount(ctx, &Account{

Owner: addr2.MarshalPB(),

Slots: []uint64{5, 6, 7, 8},

})

require.Nil(t, err)

for i, coin := range coins {

err = BridgeContract.ProcessRequestBatch(ctx, &pctypes.BridgeCurrencyRequestBatch{

Requests: []\*pctypes.BridgeCurrencyRequest{

&pctypes.BridgeCurrencyRequest{

Data: &pctypes.BridgeCurrencyRequest\_Withdraw{&pctypes.BridgeCurrencyWithdrawCoinRequest{

Owner: addr2.MarshalPB(),

Slot: coin.Slot,

}},

Meta: &pctypes.BridgeCurrencyEventMeta{

BlockNumber: uint64(i) + 1,

LogIndex: 0,

TxIndex: 0,

},

},

},

})

**BridgeAuthority Unit Test**

**func TestNextBridgeBlockNum(t \*testing.T) {**

**interval := big.NewInt(1000)**

**res := nextBridgeBlockNum(big.NewInt(9), interval)**

**assert.Equal(t, res.Cmp(big.NewInt(1000)), 0)**

**res = nextBridgeBlockNum(big.NewInt(999), interval)**

**assert.Equal(t, res.Cmp(big.NewInt(1000)), 0)**

**res = nextBridgeBlockNum(big.NewInt(0), interval)**

**assert.Equal(t, res.Cmp(big.NewInt(1000)), 0)**

**res = nextBridgeBlockNum(big.NewInt(1000), interval)**

**assert.Equal(t, res.Cmp(big.NewInt(2000)), 0)**

**res = nextBridgeBlockNum(big.NewInt(1001), interval)**

**assert.Equal(t, res.Cmp(big.NewInt(2000)), 0)**

**res = nextBridgeBlockNum(big.NewInt(1999), interval)**

**assert.Equal(t, res.Cmp(big.NewInt(2000)), 0)**

**}**

**type EthBridgeClient struct {**

**curBridgeBlockNum int64**

**BridgeChain []int64**

**allowSubmit bool**

**}**

**func (c \*EthBridgeClient) Init() error {**

**return nil**

**}**

**func (c \*EthBridgeClient) CurrentBridgeBlockNum() (\*big.Int, error) {**

**return big.NewInt(c.curBridgeBlockNum), nil**

**}**

**func (c \*EthBridgeClient) LatestEthBlockNum() (uint64, error) {**

**return 0, nil**

**}**

**func (c \*EthBridgeClient) SubmitBridgeBlock(blockNum \*big.Int, merkleRoot [32]byte) error {**

**if !c.allowSubmit {**

**return errors.New("EthBridgeClient.SubmitBridgeBlock shouldn't have been called")**

**}**

**c.BridgeChain = append(c.BridgeChain)**

**return nil**

**}**

**func (c \*EthBridgeClient) FetchDeposits(startBlock, endBlock uint64) ([]\*pctypes.BridgeDepositEvent, error) {**

**return nil, nil**

**}**

**func (c \*EthBridgeClient) FetchCoinReset(startBlock, endBlock uint64) ([]\*pctypes.BridgeCurrencyCoinResetEvent, error) {**

**return nil, nil**

**}**

**func (c \*EthBridgeClient) FetchWithdrews(startBlock, endBlock uint64) ([]\*pctypes.BridgeCurrencyWithdrewEvent, error) {**

**return nil, nil**

**}**

**func (c \*EthBridgeClient) FetchFinalizedExit(startBlock, endBlock uint64) ([]\*pctypes.BridgeCurrencyFinalizedExitEvent, error) {**

**return nil, nil**

**}**

**func (c \*EthBridgeClient) FetchStartedExit(startBlock, endBlock uint64) ([]\*pctypes.BridgeCurrencyStartedExitEvent, error) {**

**return nil, nil**

**}**

**type DAppChainBridgeClient struct {**

**curBridgeBlockNum int64**

**BridgeChain []int64**

**}**

**func (c \*DAppChainBridgeClient) Init() error {**

**return nil**

**}**

**func (c \*DAppChainBridgeClient) CurrentBridgeBlockNum() (\*big.Int, error) {**

**return big.NewInt(c.curBridgeBlockNum), nil**

**}**

**func (c \*DAppChainBridgeClient) BridgeBlockAt(blockNum \*big.Int) (\*pctypes.BridgeBlock, error) {**

**bn := blockNum.Int64()**

**for \_, b := range c.BridgeChain {**

**if b == bn {**

**return &pctypes.BridgeBlock{**

**MerkleHash: make([]byte, 32, 32),**

**}, nil**

**}**

**}**

**return nil, errors.New("block not found")**

**}**

**func (c \*DAppChainBridgeClient) FinalizeCurrentBridgeBlock() error {**

**return nil**

**}**

**func (c \*DAppChainBridgeClient) GetPendingTxs() (\*pctypes.PendingTxs, error) {**

**return nil, nil**

**}**

**func (c \*DAppChainBridgeClient) ProcessRequestBatch(requestBatch \*pctypes.BridgeCurrencyRequestBatch) error {**

**return nil**

**}**

**func (c \*DAppChainBridgeClient) GetRequestBatchTally() (\*pctypes.BridgeCurrencyRequestBatchTally, error) {**

**return nil, nil**

**}**

**func createTests() (\*EthBridgeClient, \*DAppChainBridgeClient, \*BridgeBlockWorker) {**

**ethBridgeClient := &EthBridgeClient{}**

**dappBridgeClient := &DAppChainBridgeClient{}**

**return ethBridgeClient, dappBridgeClient,**

**&BridgeBlockWorker{**

**ethBridgeClient: ethBridgeClient,**

**dappBridgeClient: dappBridgeClient,**

**BridgeBlockInterval: 1000,**

**status: BridgeBlockWorkerStatus{**

**BridgeBlockInterval: 1000,**

**},**

**}**

**}**

**func TestSyncBridgeBlocksWithEthereumWithNewChain(t \*testing.T) {**

**ethBridgeClient, dappBridgeClient, w := createTests()**

**w.Init()**

**// No blocks should be sent to Ethereum Bridge chain**

**ethBridgeClient.curBridgeBlockNum = 0**

**dappBridgeClient.curBridgeBlockNum = 0**

**if err := w.syncBridgeBlocksWithEthereum(); err != nil {**

**t.Fatal(err)**

**}**

**}**

**func TestSyncBridgeBlocksWithEthereum(t \*testing.T) {**

**ethBridgeClient, dappBridgeClient, w := createTests()**

**w.Init()**

**// TODO: setup ethBridgeClient.BridgeChain & dappBridgeClient.BridgeChain**

**ethBridgeClient.curBridgeBlockNum = 0**

**dappBridgeClient.curBridgeBlockNum = 0**

**if err := w.syncBridgeBlocksWithEthereum(); err != nil {**

**t.Fatal(err)**

**}**

**}**

BridgeWorker Protocol Buffers

enum BridgeCurrencyCoinState {

DEPOSITED = 0;

EXITING = 1;

CHALLENGED = 2;

EXITED = 3;

}

// Bridge Currency coin holds a single ERC721 token

message BridgeCurrencyCoin {

// Unique ID

uint64 slot = 1 [jstype = JS\_STRING];

BridgeCurrencyCoinState state = 2;

// ERC721 token ID

BigUInt token = 3;

// ERC721 token contract address

Address contract = 4;

}

message BridgeCurrencyAccount {

Address owner = 1;

// Bridge coins in this account, identified by their slot number.

repeated uint64 slots = 3 [jstype = JS\_STRING];

}

message BridgeBlock {

BigUInt uid = 1;

repeated BridgeTx transactions = 2; // List of transactions included in Block

bytes signature = 3; // Signature on the block’s hash

bytes merkle\_hash = 4; // The block's merkle root from its included transactions

bytes hash = 5; // The hash of the RLP encoded unsigned block’s bytes.

bytes proof = 6;

}

message BridgeTx {

uint64 slot = 1 [jstype = JS\_STRING]; // The slot of the UTXO - Currently uint64, subject to change.

BigUInt previous\_block = 2 [(gogoproto.customname)="PreviousBlock"]; //Each time a transaction is created, it MUST refer to a previous block which also included that transaction. A transaction is considered a “deposit transaction”, if it’s the first UTXO after a user deposits their coin in the Bridge Chain. This transaction mints coins from nowhere in the Bridge Chain and as a result its previous block is 0.

BigUInt denomination = 3; // How many coins are included in that UTXO. Currently this is always 1 since we’re using ERC721 tokens which are unique, however in future iterations this can be any number.

Address new\_owner = 4 [(gogoproto.customname)="NewOwner"]; // The new owner of the transaction.

bytes signature = 5; // Signature on the transaction's hash

bytes hash = 6; // The hash of the RLP encoded unsigned transaction’s bytes. If the transaction is a deposit transaction (its prevblock is 0), its hash is the hash of its uid

bytes merkle\_hash = 7; // The hash of the RLP encoded signed transaction’s bytes

Address sender = 8; // The transaction’s sender, derived from the hash and the signature

bytes proof = 9; // Proof after its inserted into the block

}

message BridgeBookKeeping {

BigUInt current\_height = 1 [(gogoproto.customname)="CurrentHeight"]; // Height of all the Bridge blocks

}

message GetCurrentBlockRequest {

}

message GetCurrentBlockResponse {

BigUInt block\_height = 1 [(gogoproto.customname)="BlockHeight"];

}

message GetBlockRequest {

BigUInt block\_height = 1 [(gogoproto.customname)="BlockHeight"];

}

message GetBlockResponse {

BridgeBlock block = 1;

}

// This only originates from the validator

message SubmitBlockToMainnetRequest {

}

message SubmitBlockToMainnetResponse {

bytes merkle\_hash = 1; // Merkle root of Bridge block to be submitted to mainnet

}

message BridgeTxRequest {

BridgeTx Bridgetx = 1;

}

message BridgeTxResponse {

}

message GetBridgeTxRequest {

uint64 slot = 1 [jstype = JS\_STRING]; // The slot of the UTXO - Currently uint64, subject to change.

BigUInt block\_height = 2 [(gogoproto.customname)="BlockHeight"];

}

message GetBridgeTxResponse {

BridgeTx Bridgetx = 1;

}

message GetUserSlotsRequest {

Address from = 1;

}

message GetUserSlotsResponse {

repeated uint64 slots = 1 [jstype = JS\_STRING];

}

// This only originates from the validator

message DepositRequest {

uint64 slot = 1 [jstype = JS\_STRING]; // The slot of the UTXO - Currently uint64, subject to change.

BigUInt deposit\_block = 2;

// For ERC20 this is the number of coins deposited, for ERC721 this is a token ID.

BigUInt denomination = 3;

// Entity that made the deposit

Address from = 4;

// Contract from which the coins originated (i.e. the currency of the coins)

Address contract = 5;

}

message DepositResponse {

}

message BridgeCurrencyCoinResetRequest {

Address owner = 1;

uint64 slot = 2 [jstype = JS\_STRING];

}

message BridgeCurrencyExitCoinRequest {

Address owner = 1;

uint64 slot = 2 [jstype = JS\_STRING];

}

message BridgeCurrencyWithdrawCoinRequest {

Address owner = 1;

uint64 slot = 2 [jstype = JS\_STRING];

}

message BridgeCurrencyRequest {

oneof data {

DepositRequest deposit = 1;

BridgeCurrencyCoinResetRequest coin\_reset = 2;

BridgeCurrencyExitCoinRequest started\_exit = 3;

BridgeCurrencyWithdrawCoinRequest withdraw = 4;

}

BridgeCurrencyEventMeta meta = 5;

}

message BridgeCurrencyRequestBatch {

repeated BridgeCurrencyRequest requests = 1;

}

message GetPendingTxsRequest{

}

message PendingTxs {

repeated BridgeTx transactions = 1;

}

// Initialization of state from Genesis.json

message BridgeCurrencyParams {

uint64 block\_interval = 1;

}

message BridgeCurrencyInitRequest {

BridgeCurrencyParams params = 1;

Address BridgeAuthority = 2;

}

// Static Calls

message BridgeCurrencyBalanceOfRequest {

Address owner = 1;

Address contract = 2;

}

message BridgeCurrencyBalanceOfResponse {

repeated BridgeCurrencyCoin coins = 1;

}

message BridgeCurrencyUpdateBridgeAuthorityRequest {

Address new\_BridgeAuthority = 1;

}

// Event structures

message BridgeCurrencyRequestBatchTally {

uint64 last\_seen\_block\_number = 1;

uint64 last\_seen\_tx\_index = 2;

uint64 last\_seen\_log\_index = 3;

}

message BridgeCurrencyGetRequestBatchTallyRequest {

}

message BridgeCurrencyEventMeta {

uint64 block\_number = 1;

uint64 tx\_index = 2;

uint64 log\_index = 3;

}

message BridgeDepositEvent {

uint64 slot = 1; // The slot of the UTXO - Currently uint64, subject to change.

BigUInt deposit\_block = 2;

// For ERC20 this is the number of coins deposited, for ERC721 this is a token ID.

BigUInt denomination = 3;

// Entity that made the deposit

Address from = 4;

// Contract from which the coins originated (i.e. the currency of the coins)

Address contract = 5;

BridgeCurrencyEventMeta meta = 6;

}

message BridgeCurrencyCoinResetEvent {

Address owner = 1;

uint64 slot = 2 [jstype = JS\_STRING];

BridgeCurrencyEventMeta meta = 3;

}

message BridgeCurrencyStartedExitEvent {

Address owner = 1;

uint64 slot = 2 [jstype = JS\_STRING];

BridgeCurrencyEventMeta meta = 3;

}

message BridgeCurrencyFinalizedExitEvent {

Address owner = 1;

uint64 slot = 2 [jstype = JS\_STRING];

BridgeCurrencyEventMeta meta = 3;

}

message BridgeCurrencyWithdrewEvent {

Address owner = 1;

uint32 mode = 2;

Address contract = 3;

BigUInt uid = 4;

BigUInt denomination = 5;

uint64 slot = 6 [jstype = JS\_STRING];

BridgeCurrencyEventMeta meta = 7;

}

// Events emitted by the contract

message BridgeCurrencyTransferConfirmed {

Address from = 1;

Address to = 2;

uint64 slot = 3 [jstype = JS\_STRING];

}

message BridgeCurrencyExitConfirmedEvent {

Address owner = 1;

uint64 slot = 2 [jstype = JS\_STRING];

}

message BridgeCurrencyResetConfirmedEvent {

Address owner = 1;

uint64 slot = 2 [jstype = JS\_STRING];

}

message BridgeCurrencyWithdrawConfirmedEvent {

BridgeCurrencyCoin coin = 1;

Address owner = 2;

uint64 slot = 3 [jstype = JS\_STRING];

}

message BridgeCurrencyDepositConfirmedEvent {

BridgeCurrencyCoin coin = 1;

Address owner = 2;

}

message BridgeCurrencySubmitBlockConfirmedEvent {

uint64 number\_of\_pending\_transactions = 1;

BigUInt current\_block\_height = 2;

bytes merkle\_hash = 3;

}