type (

InitRequest = pctypes.PlasmaCashInitRequest

SubmitBlockToMainnetRequest = pctypes.SubmitBlockToMainnetRequest

SubmitBlockToMainnetResponse = pctypes.SubmitBlockToMainnetResponse

GetBlockRequest = pctypes.GetBlockRequest

GetBlockResponse = pctypes.GetBlockResponse

PlasmaTxRequest = pctypes.PlasmaTxRequest

PlasmaTxResponse = pctypes.PlasmaTxResponse

DepositRequest = pctypes.DepositRequest

PlasmaTx = pctypes.PlasmaTx

GetCurrentBlockResponse = pctypes.GetCurrentBlockResponse

GetCurrentBlockRequest = pctypes.GetCurrentBlockRequest

PlasmaBookKeeping = pctypes.PlasmaBookKeeping

PlasmaBlock = pctypes.PlasmaBlock

PendingTxs = pctypes.PendingTxs

CoinState = pctypes.PlasmaCashCoinState

Coin = pctypes.PlasmaCashCoin

Account = pctypes.PlasmaCashAccount

BalanceOfRequest = pctypes.PlasmaCashBalanceOfRequest

BalanceOfResponse = pctypes.PlasmaCashBalanceOfResponse

CoinResetRequest = pctypes.PlasmaCashCoinResetRequest

ExitCoinRequest = pctypes.PlasmaCashExitCoinRequest

WithdrawCoinRequest = pctypes.PlasmaCashWithdrawCoinRequest

TransferConfirmed = pctypes.PlasmaCashTransferConfirmed

GetPlasmaTxRequest = pctypes.GetPlasmaTxRequest

GetPlasmaTxResponse = pctypes.GetPlasmaTxResponse

GetUserSlotsRequest = pctypes.GetUserSlotsRequest

GetUserSlotsResponse = pctypes.GetUserSlotsResponse

UpdateOracleRequest = pctypes.PlasmaCashUpdateOracleRequest

GetPendingTxsRequest = pctypes.GetPendingTxsRequest

RequestBatchTally = pctypes.PlasmaCashRequestBatchTally

GetRequestBatchTallyRequest = pctypes.PlasmaCashGetRequestBatchTallyRequest

)

const (

CoinState\_DEPOSITED = pctypes.PlasmaCashCoinState\_DEPOSITED

CoinState\_EXITING = pctypes.PlasmaCashCoinState\_EXITING

CoinState\_CHALLENGED = pctypes.PlasmaCashCoinState\_CHALLENGED

CoinState\_EXITED = pctypes.PlasmaCashCoinState\_EXITED

contractPlasmaCashTransferConfirmedEventTopic = "event:PlasmaCashTransferConfirmed"

oracleRole = "pcash\_role\_oracle"

addressMapperContractName = "addressmapper"

)

type PlasmaCash struct {

}

var (

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

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

accountKeyPrefix = []byte("account")

plasmaMerkleTopic = "pcash\_mainnet\_merkle"

SubmitBlockConfirmedEventTopic = "pcash:submitblockconfirmed"

ExitConfirmedEventTopic = "pcash:exitconfirmed"

WithdrawConfirmedEventTopic = "pcash:withdrawconfirmed"

ResetConfirmedEventTopic = "pcash:resetconfirmed"

DepositConfirmedEventTopic = "pcash:depositconfirmed"

ChangeOraclePermission = []byte("change\_oracle")

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

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

)

func accountKey(addr panthersidechain.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("pcash\_block\_"), []byte(height.String()))

}

func requestBatchTallyKey() []byte {

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

}

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

return plugin.Meta{

Name: "plasmacash",

Version: "1.0.0",

}, nil

}

func (c \*PlasmaCash) 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 \*PlasmaCash) 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 \*PlasmaCash) registerOracle(ctx contract.Context, pbOracle \*types.Address, currentOracle \*panthersidechain.Address) error {

if pbOracle == nil {

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

}

newOracleAddr := panthersidechain.UnmarshalAddressPB(pbOracle)

if newOracleAddr.IsEmpty() {

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

}

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

if currentOracle != nil {

ctx.RevokePermissionFrom(\*currentOracle, ChangeOraclePermission, oracleRole)

ctx.RevokePermissionFrom(\*currentOracle, SubmitEventsPermission, oracleRole)

}

ctx.GrantPermissionTo(newOracleAddr, ChangeOraclePermission, oracleRole)

ctx.GrantPermissionTo(newOracleAddr, SubmitEventsPermission, oracleRole)

return nil

}

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

//params := req.Params

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

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

}

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

Value: \*panthersidechain.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 \*PlasmaCash) UpdateOracle(ctx contract.Context, req \*UpdateOracleRequest) error {

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

return ErrNotAuthorized

}

currentOracle := ctx.Message().Sender

return c.registerOracle(ctx, req.NewOracle, &currentOracle)

}

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

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

NumberOfPendingTransactions: uint64(numPendingTransactions),

CurrentBlockHeight: blockHeight,

MerkleHash: merkleHash,

})

if err != nil {

return err

}

ctx.EmitTopics(marshalled, SubmitBlockConfirmedEventTopic)

return nil

}

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

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

Owner: owner,

Slot: slot,

})

if err != nil {

return err

}

ctx.EmitTopics(marshalled, ExitConfirmedEventTopic)

return nil

}

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

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

Coin: coin,

Owner: owner,

Slot: slot,

})

if err != nil {

return err

}

ctx.EmitTopics(marshalled, WithdrawConfirmedEventTopic)

return nil

}

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

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

Owner: owner,

Slot: slot,

})

if err != nil {

return err

}

ctx.EmitTopics(marshalled, ResetConfirmedEventTopic)

return nil

}

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

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

Coin: coin,

Owner: owner,

})

if err != nil {

return err

}

ctx.EmitTopics(marshalled, DepositConfirmedEventTopic)

return nil

}

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

//TODO prevent this being called to oftern

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

//Raise blockheight

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

return nil, ErrNotAuthorized

}

pbk := &PlasmaBookKeeping{}

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")

//TODO do this rounding in a bigint safe way

// round to nearest 1000

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

pbk.CurrentHeight.Value = \*panthersidechain.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 := &PlasmaBlock{

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, plasmaMerkleTopic)

//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 \*PlasmaCash) verifyPlasmaRequest(ctx contract.Context, req \*PlasmaTxRequest) error {

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

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

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

}

claimedSender := panthersidechain.UnmarshalAddressPB(req.Plasmatx.Sender)

panthersidechainTx := &plasma\_cash.PanthersidechainTx{

Slot: req.Plasmatx.Slot,

Denomination: req.Plasmatx.Denomination.Value.Int,

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

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

TXProof: req.Plasmatx.Proof,

}

calculatedPlasmaTxHash, err := panthersidechainTx.Hash()

if err != nil {

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

}

senderEthAddressFromPlasmaSig, err := evmcompat.RecoverAddressFromTypedSig(

calculatedPlasmaTxHash, req.Plasmatx.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 plasmatx 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(senderEthAddressFromPlasmaSig.Bytes(), claimedSender.Local) != 0 ||

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

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

}

return nil

}

func (c \*PlasmaCash) PlasmaTxRequest(ctx contract.Context, req \*PlasmaTxRequest) error {

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

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

return ErrNotAuthorized

}

sender := panthersidechain.UnmarshalAddressPB(req.Plasmatx.Sender)

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

pending := &PendingTxs{}

ctx.Get(pendingTXsKey, pending)

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

if v.Slot == req.Plasmatx.Slot {

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

}

}

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

receiver := panthersidechain.UnmarshalAddressPB(req.Plasmatx.NewOwner)

coin, err := loadCoin(ctx, req.Plasmatx.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 \*PlasmaCash) depositRequest(ctx contract.Context, req \*DepositRequest) error {

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

pbk := &PlasmaBookKeeping{}

ctx.Get(blockHeightKey, pbk)

pending := &PendingTxs{}

ctx.Get(pendingTXsKey, pending)

// create a new deposit block for the deposit event

tx := &PlasmaTx{

Slot: req.Slot,

Denomination: req.Denomination,

NewOwner: req.From,

Proof: make([]byte, 8),

}

pb := &PlasmaBlock{

//MerkleHash: merkleHash,

Transactions: []\*PlasmaTx{tx},

Uid: req.DepositBlock,

}

//TODO 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 := "[PlasmaCash] failed to process deposit"

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

ownerAddr := panthersidechain.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 Plasma coins owned by an entity. The request must specify the address of

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

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

ownerAddr := panthersidechain.UnmarshalAddressPB(req.Owner)

account, err := loadAccount(ctx, ownerAddr)

if err != nil {

return nil, errors.Wrap(err, "[PlasmaCash] 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 Plasma coin from EXITING to DEPOSITED

// This method should only be called by the Plasma Cash Oracle when a coin's exit is successfully challenged

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

defaultErrMsg := "[PlasmaCash] 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("[PlasmaCash] 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 Plasma coin from DEPOSITED to EXITING.

// This method should only be called by the Plasma Cash Oracle when it detects an attempted exit

// of a Plasma coin on Ethereum Mainnet.

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

defaultErrMsg := "[PlasmaCash] 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("[PlasmaCash] 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 Plasma coin from a local Plasma account.

// This method should only be called by the Plasma Cash Oracle when it detects a withdrawal of a

// Plasma coin on Ethereum Mainnet.

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

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

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

if err != nil {

return errors.Wrap(err, defaultErrMsg)

}

ownerAddr := panthersidechain.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 \*PlasmaCash) GetCurrentBlockRequest(ctx contract.StaticContext, req \*GetCurrentBlockRequest) (\*GetCurrentBlockResponse, error) {

pbk := &PlasmaBookKeeping{}

ctx.Get(blockHeightKey, pbk)

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

}

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

pb := &PlasmaBlock{}

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

if err != nil {

return nil, err

}

return &GetBlockResponse{Block: pb}, nil

}

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

if req.From == nil {

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

}

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

if err != nil {

return nil, err

}

res := &GetUserSlotsResponse{}

res.Slots = reqAcct.Slots

return res, nil

}

func (c \*PlasmaCash) GetPlasmaTxRequest(ctx contract.StaticContext, req \*GetPlasmaTxRequest) (\*GetPlasmaTxResponse, error) {

pb := &PlasmaBlock{}

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 := &PlasmaTx{}

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 := &GetPlasmaTxResponse{

Plasmatx: tx,

}

return res, nil

}

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

if hasPermission, \_ := ctx.HasPermission(SubmitEventsPermission, []string{oracleRole}); !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.PlasmaCashRequest\_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.PlasmaCashRequest\_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.PlasmaCashRequest\_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.PlasmaCashRequest\_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 panthersidechain.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 := panthersidechain.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 Plasma accounts to reflect a Plasma coin transfer.

func transferCoin(ctx contract.Context, coin \*Coin, sender, receiver panthersidechain.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, contractPlasmaCashTransferConfirmedEventTopic)

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 \*PlasmaTx) ([]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.PlasmaCashEventMeta, 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 \*PlasmaTx) ([]byte, error) {

return rlp.EncodeToBytes([]interface{}{

uint64(pb.Slot),

pb.PreviousBlock.Value.Bytes(),

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

pb.GetNewOwner().Local,

})

}