6.3. CSC and CSU Descriptions - Computer Software Components (CSC) and Computer Software Units (CSU)

6.3.1 Class Descriptions

6.3.1.1 Peer-2-peer network

The peer-2-peer network is the key component of how the individual nodes communicate with each other. From a high level perspective, each valid node in a decentralized system will act as a peer in the system. Each node will have two streams for each unique node in the system. One of these streams will be dedicated to sending data from the current node to the other node that the stream connects to. The other stream is used for receiving data from the other node into the current node. By having the network configured this way, there is no singular node that is required for having the decentralized system running. Each node is equally important as the others. This allows nodes to be able to enter or disconnect from the system at their leisure.

6.3.1.2 Blockchain

The blockchain is the structure used for storing transactions that happen on the decentralized database. Blocks are created with transactions, a public key, a previous public key, and an index. The blockchain contains methods for block creation, adding blocks to the blockchain, and validating block information with the latest block on the blockchain.

6.3.1.3 Proof of Stake

Proof of Stake(PoS) revolves around how to update node states given information from the debased system and how to inform the debased system as needed. This includes launching routines at the behest of the system for the voting process.

6.3.2 Detailed Interface Descriptions

6.3.2.1 Peer-2-peer

6.3.2.1.1 Host

The host is a component that functions as the primary identifier for the node. When instantiating the host, the node creates an address and identity for the node itself. If other nodes wish to connect or communicate with the current node, then they must use the generated address and have the proper credentials to establish a stream. Essentially, the host acts as the primary identity handler for any incoming/outgoing information

6.3.2.1.2 Streams

Streams function as the direct connection between any two nodes. After a stream is initially set up, it has the ability to connect to another node. With this connection, the stream can be used for either sending or receiving data over from one node to the other. Some of the fields included here are the protocol being used by the stream and the connection field itself.

6.3.2.1.3 Reader/Writer

The reader/writer are used for handling the data itself over a properly configured stream. The two fields of the reader/writer are the reader and writer. The reader is specifically used for reading data that has been added to the reader. The writer is used for transferring data that has been added to the writer. When wanting to access/manipulate the data, the node simple needs to request permission from the reader and writer to communicate with it.

6.3.2.1.4 Main function

The main function is used for creating a new node and connecting that new node to other nodes within the decentralized system.

6.3.2.1.5 Stream Handler

The stream handler is used for handling incoming streams that are coming from a new node. This will determine what to do with the incoming stream and spawn new reader and writer functions if needed.

6.3.2.1.6 Read incoming data

This function is used for reading incoming data that is to be received by a node. When no data is being received, the read function will wait until data is being sent over.

6.3.2.1.7 Write outgoing data

This function is being used for writing data from a node to a stream that it is hooked up to. When there is nothing being written, the function will wait until the node requests to.

6.3.3 Detailed Data Structure Descriptions

6.3.3.1 Peer-2-peer

6.3.3.1.1 Main handler to read/write handlers

The main handler sets up the initial stream and host id. By doing this, the new node now has an ID and a stream that can be used to communicate to a singular node. This stream is now passed onto both the reader and writer functions in a goroutine. Since the stream is now present in both of these functions, other nodes are able to communicate with the current node and the current node is able to communicate to other nodes.

6.3.3.1.2 Stream handler to reader/writer handler

Whenever a new node is trying to connect to another node, the new node will trigger the other node's stream handler function. After setting up a new stream for the new node and the other node to communicate with, the stream handler will pass this stream onto the reader and writer functions in another goroutine. By having these new reader and writer functions running in a goroutine, the new node is now able to communicate bidirectionally with the other node.

6.3.3.1.3 Dropping a node from the decentralized system

Since every node can act as an entry point for the decentralized system, handling a dropped node is fairly straightforward. When a node disconnects, whether elegantly or suddenly, the

other nodes will terminate the stream that was being used to communicate with the disconnected node. The stream is then removed from the stream's map and the respective goroutines are terminated as well. By doing this, all of the nodes in the decentralized system have removed the existence of this stream safely and can continue communicating with the other nodes.

6.3.3.2 Blockchain

6.3.3.2.1 Creating a block

Blocks are given a public key, the previous block's public key, a list of transactions, and an index at creation. A created block can then be added to the blockchain. A created block is not automatically added to the blockchain at creation.

6.3.3.2.2 Calculating the hash of a block

A SHA256 hash is generated for each block at block creation and acts as the block's public key. The string used to compute the SHA256 hash which is the public key of the block is a joined combination of the index, transactions, and previous public key respectively.

6.3.3.2.3 Validating the public key and index of a block on the blockchain

The blockchain has a method to validate consecutive blocks in the blockchain to ensure that a certain block's previous public key matches up with the public key of the block before it. The method also ensures that the index of any new block is valid by comparing it to the previous block's index. This method can be used to validate two consecutive blocks on the blockchain. Lastly, the method recalculates the hash of the block and checks it with the existing public key of the block.

6.3.3.2.4 Adding a block to the blockchain

The blockchain has a method for adding blocks to it. This essentially appends a pointer to the list of pointers to blocks on the back-end. A block is not added to the blockchain at creation. A block is added to the blockchain after validation.

6.3.3.3 Proof of Stake

6.3.3.3.1 Block Generation

The node can place a bid for generating the next block. This message is sent to the other nodes on the network and includes (bid price, stake, block number, est. time to create block, signature). Each node records the bids placed, when a bid is received by a node it is stored in an unordered list corresponding to the work being bid on. The node that created the best bid as chosen by the shared bid choosing algorithm, creates the block as detailed in the blockchain section and sends it to the network including (block, updated metadata, signature).

6.3.3.3.2 Block Validation

This process is started upon receiving a newly generated block. The nodes' voting time slot is determined using the node ordering protocol and assigning a voting window. Each node can determine every voting window locally. When a vote is received it is recorded and tallied if received within the nodes' window. A vote is created by a node by checking the block received follows the block generation rules as outlined in the blockchain section and by being composed of only pending transactions that are correctly chosen and transcribed; the metadata must also be correctly updated in accordance with the transactions.

6.3.3.3.3 Create Transaction

This process is started when a transaction is started locally on the node through the command line interface. The node attempts to fulfill the request locally, but if it requires data not present or manipulation of the blockchain state, the transaction is sent to the network as a pending transaction. The transactions are as listed (each transaction ends with the signature of the origin node):

CreateAcct(publicAcct#, originNodeSignature)

MoveFunds(senderAcct#, receiverAcct#, senderSignature, ONS)

CreateTable(tableMetadata, maxCostAllowed, ownerAcct#, ownerSign, ONS)

AddData(tagetTable, maxCostAllowed, manipulatorAcct#, data, manipulatorSignature, ONS)

ReadData(tagetTable, SQL, maxCostAllowed, readerAcct, ONS)

EditData(tagetTable, SQL, maxCostAllowed, manipulatorAcct#, data, manipulatorSignature, ONS)

DeleteData(tagetTable, SQL, maxCostAllowed, manipulatorAcct#, manipulatorSignature, ONS)

DeleteTable(tagetTable, maxCostAllowed, manipulatorAcct#, manipulatorSignature, ONS)

UpdatePermissions([((ability, tableId), acct),], maxCostAllowed, authorizingAcct, AuthorSign, ONS)

CheckAcctBalance(acctId, maxCostAllowed, acctOwnerSign, ONS)

CheckPermissions(acctId, maxCostAllowed, acctOwnerSign, ONS)

CheckTableMetadata(tagetTable, SQL, maxCostAllowed, readerAcct#, data, readerSignature, ONS)

CheckTableHistory(tagetTable, SQL, maxCostAllowed, readerSignature, ONS)

CheckAcctBalanceHistory(acctId, maxCostAllowed, readerAcct#, data, readerSignature, ONS)

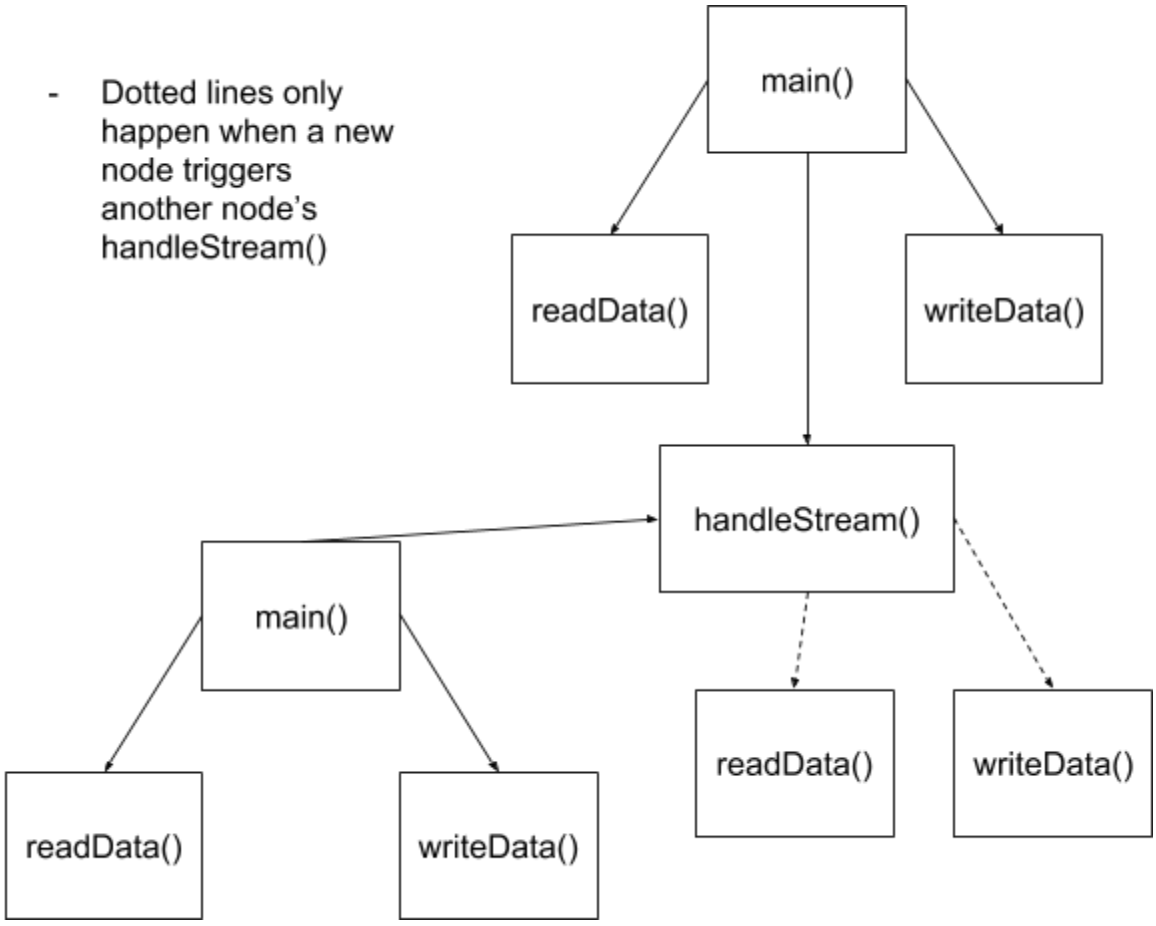
6.3.3.3.4 Data Retrieval

When the node requires data that is not stored locally, it sends a request to the network for said data. requestData([hash1,... , hashn], maxCostAllowed, requestingAcct, requestingAcctSignature, ONS). When a data request is received, it is added to the list of pending requests. The request fulfillment bidding system begins, following the same bidding process as before. The bid winner sends the corresponding data (data, request#, fulfillerNode, fulfillerSignature) to the

system for verification. The system votes in the same process detailed above for blocks. Once the data is verified it is sent to the origin node.

6.3.4 Detailed Design Diagrams

6.3.4.1 P2P case diagram

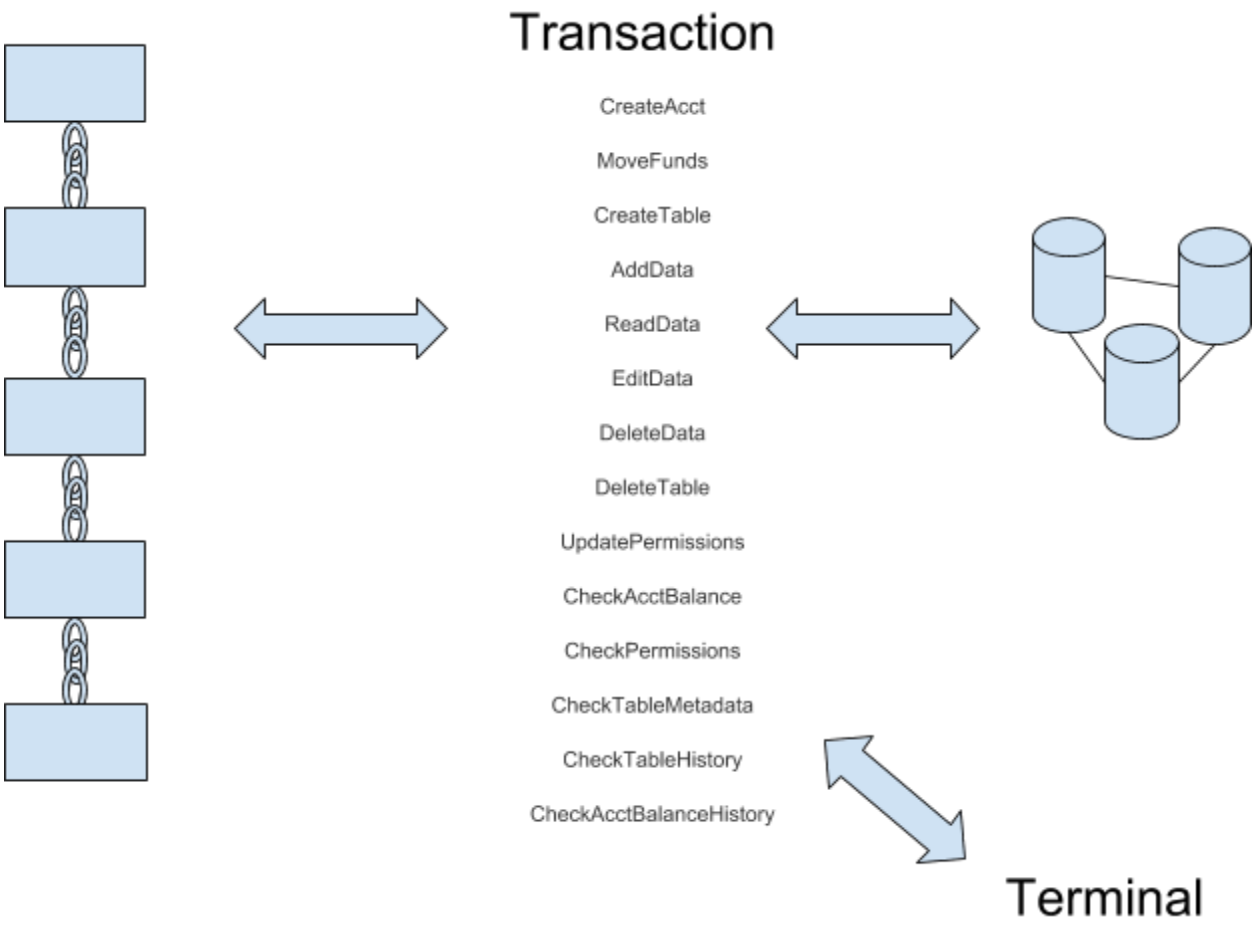


6.3.4.2 P2P package diagram

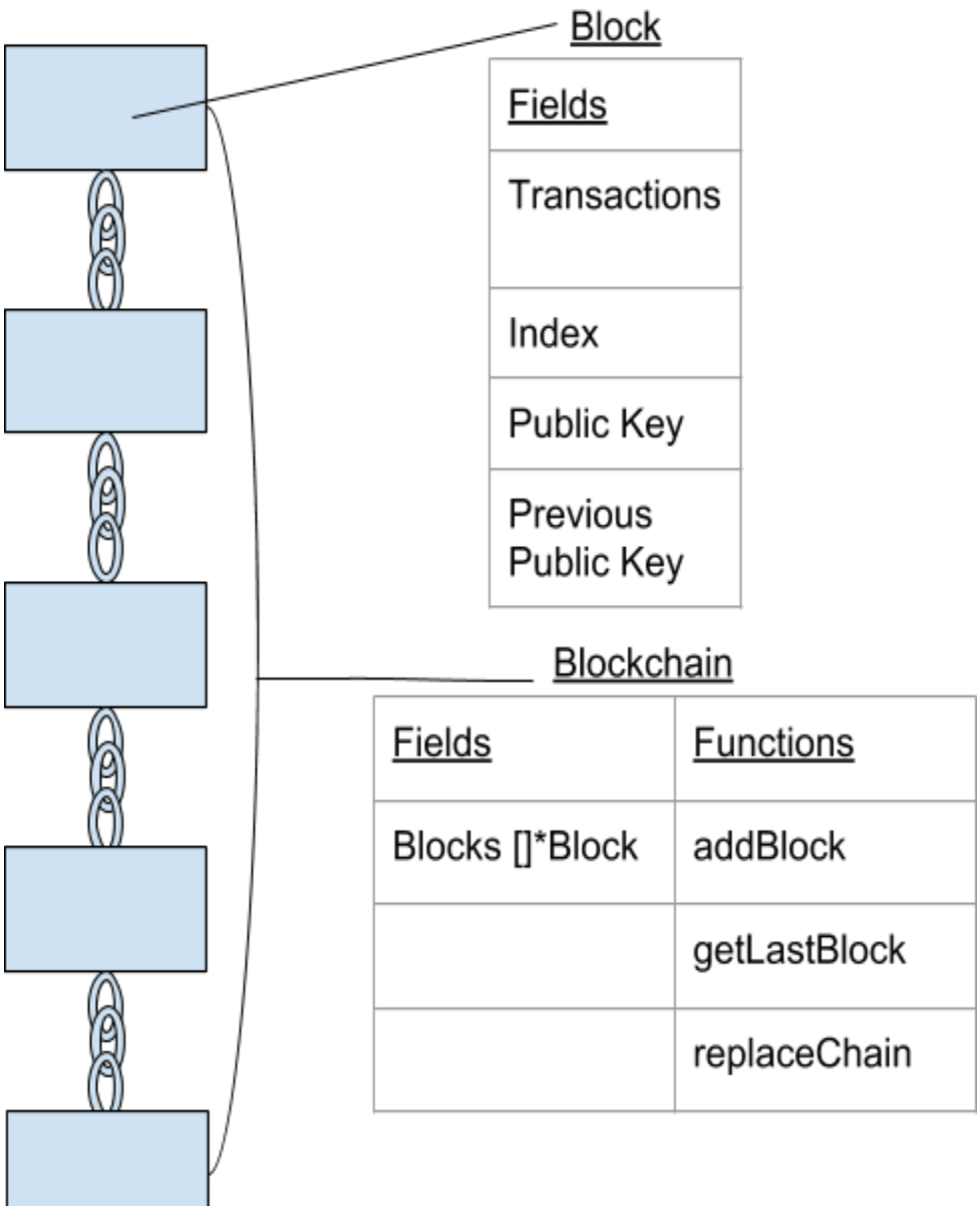
peer2peer.go

```
"bufio"
"context"
"crypto/rand"
"encoding/gob"
"encoding/json"
"flag"
"fmt"
"io"
"log"
mrand "math/rand"
"os"

"github.com/libp2p/go-libp2p"
swarm "github.com/libp2p/go-libp2p-swarm"

"github.com/libp2p/go-libp2p-crypto"
"github.com/libp2p/go-libp2p-host"
"github.com/libp2p/go-libp2p-net"
"github.com/libp2p/go-libp2p-peer"
"github.com/libp2p/go-libp2p-peerstore"
"github.com/multiformats/go-multiaddr"
```

6.3.5.1 PoS case diagram


Transaction
CreateAcct
MoveFunds
CreateTable
AddData
ReadData
EditData
DeleteData
DeleteTable
UpdatePermissions
CheckAcctBalance
CheckPermissions
CheckTableMetadata
CheckTableHistory
CheckAcctBalanceHistory
Terminal

6.3.6.1 Blockchain diagram



6.3.6.1 Blockchain package diagram

blockchain.go

```
“crypto/sha256”
“encoding/hex”
“log”
“strings”
"github.com/davecgh/go-spew/spew"
```