Project Report

i16-0100 – Amer Farooq

i16-0005 – Faiq Iftikhar

i16-0077 – Zulkifl Rasheed

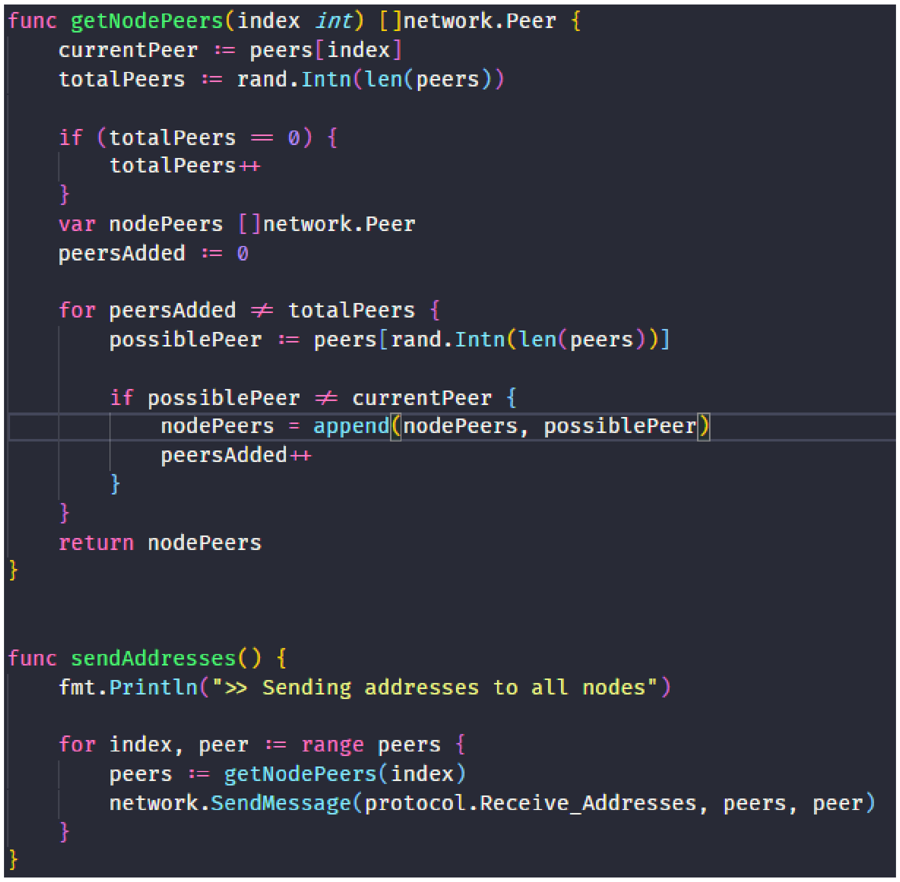
Consensus Algorithm

After going over a variety of ideas and considering the merits of existing consensus algorithms, we decided to go for a variation of the Proof-of-Stake algorithm. In our case, we wanted to reward users that had a high throughput of transactions as well as a high stake. Hence in our algorithm, the stake and the user's transaction activity are both equally important in deciding who will create the next block.

One major assumption that we have made is the existence of a central program called the director. We envisioned this as a sort of contract that is part of the blockchain and handles the logic related to choosing the next block miner.

Algorithm Explanation

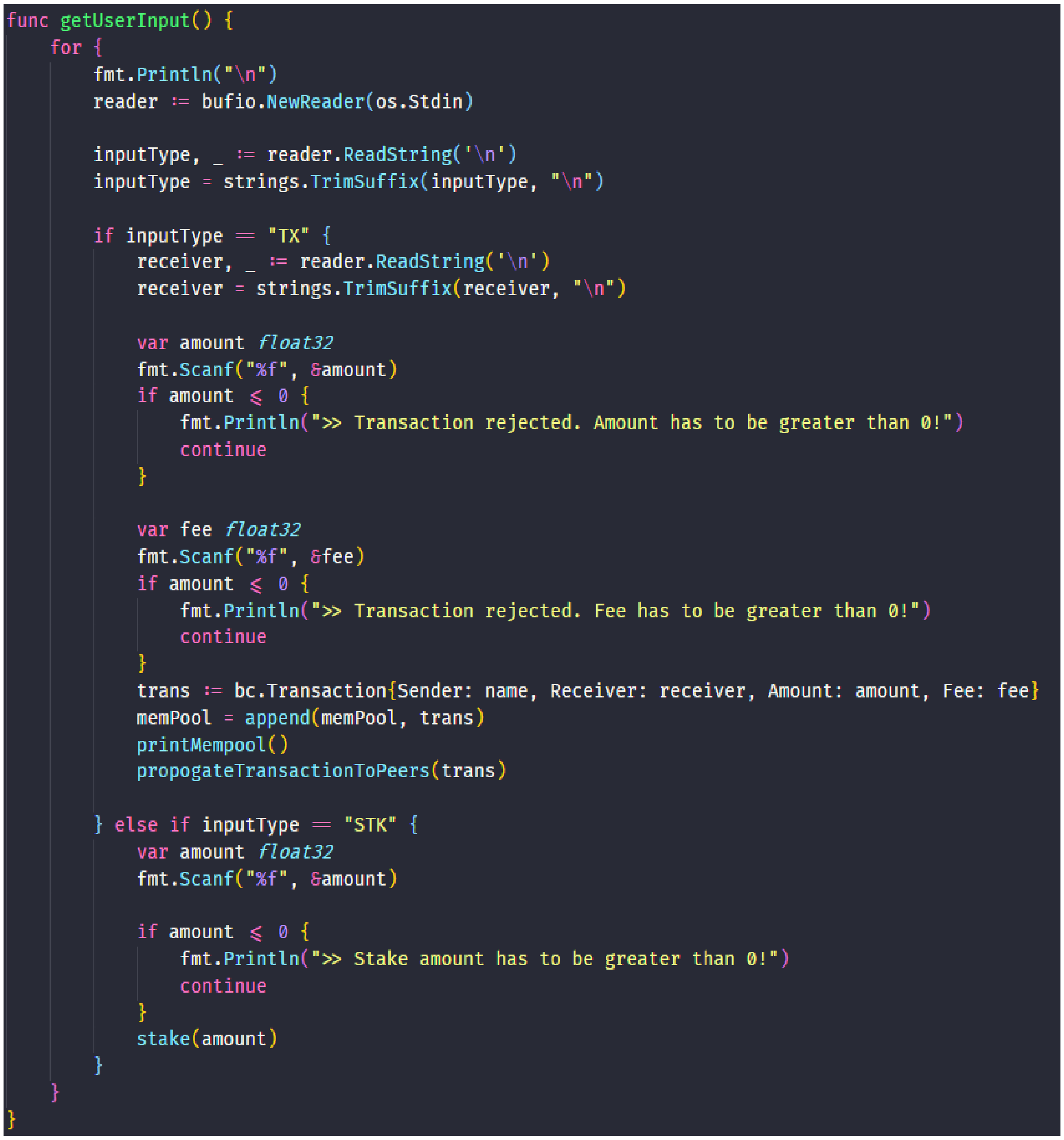
1. A **network manager** is first started that is responsible for creating connections between the nodes that join the network. Once a certain number of nodes have connected to the manager, its creates random connections between these nodes and sends them their peers.



```go
func getNodePeers(index int) []network.Peer {
    currentPeer := peers[index]
    totalPeers := rand.Intn(len(peers))

    if (totalPeers == 0) {
        totalPeers++
    }
    var nodePeers []network.Peer
    peersAdded := 0

    for peersAdded != totalPeers {
        possiblePeer := peers[rand.Intn(len(peers))]

        if possiblePeer != currentPeer {
            nodePeers = append(nodePeers, possiblePeer)
            peersAdded++
        }
    }
    return nodePeers
}


func sendAddresses() {
    fmt.Println(">> Sending addresses to all nodes")

    for index, peer := range peers {
        peers := getNodePeers(index)
        network.SendMessage(protocol.Receive_Addresses, peers, peer)
    }
}
```

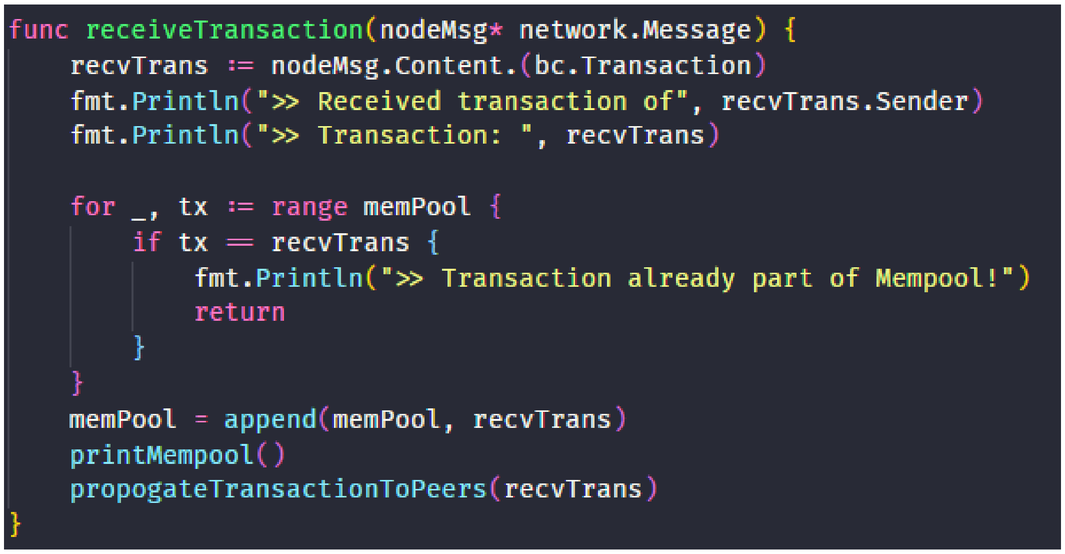
2. The network manager also asks the first node that connects to mine the genesis block. When all the nodes have connected and received their peers, the node that mined the genesis block floods it into the network and all other nodes add it to their blockchain.
3. After starting the manager, the **director** is started. This program implements the main logic and is part of the blockchain.

4. The nodes than begin waiting for user input. The user can send a transaction, specifying the **receiver,** the **amount** and a **transaction fee.**



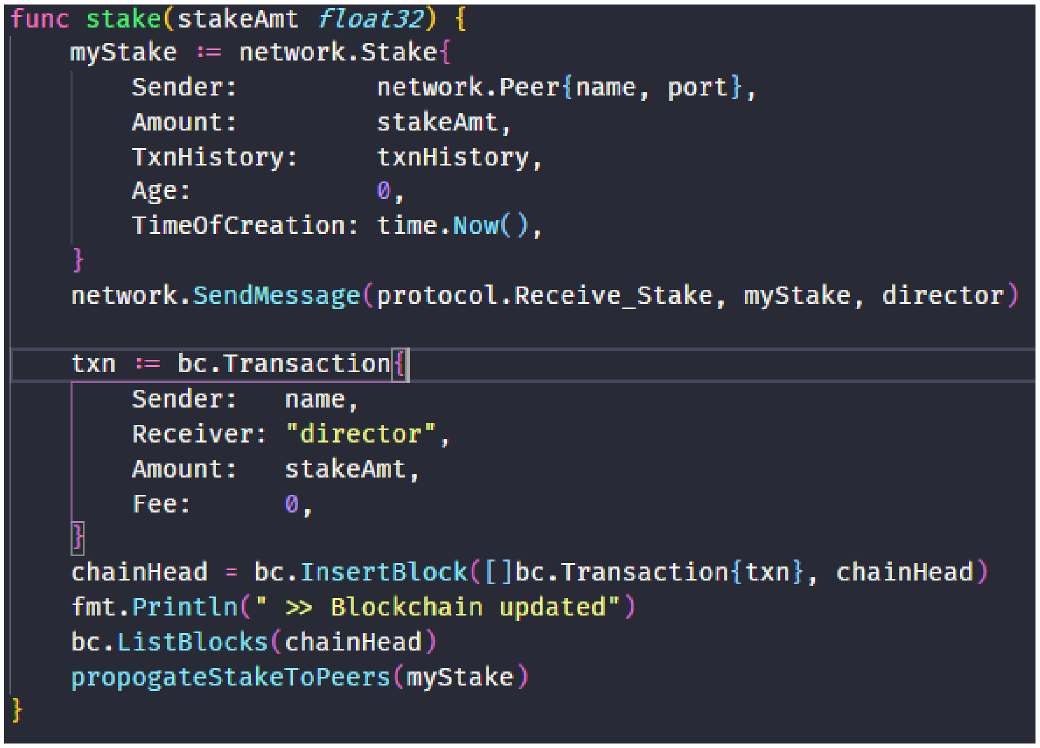
```go
func getUserInput() {
    for {
        fmt.Println("\n")
        reader := bufio.NewReader(os.Stdin)

        inputType, _ := reader.ReadString('\n')
        inputType = strings.TrimSuffix(inputType, "\n")

        if inputType == "TX" {
            receiver, _ := reader.ReadString('\n')
            receiver = strings.TrimSuffix(receiver, "\n")

            var amount float32
            fmt.Scanf("%f", &amount)
            if amount <= 0 {
                fmt.Println(">> Transaction rejected. Amount has to be greater than 0!")
                continue
            }

            var fee float32
            fmt.Scanf("%f", &fee)
            if amount <= 0 {
                fmt.Println(">> Transaction rejected. Fee has to be greater than 0!")
                continue
            }
            trans := bc.Transaction{Sender: name, Receiver: receiver, Amount: amount, Fee: fee}
            memPool = append(memPool, trans)
            printMempool()
            propogateTransactionToPeers(trans)

        } else if inputType == "STK" {
            var amount float32
            fmt.Scanf("%f", &amount)

            if amount <= 0 {
                fmt.Println(">> Stake amount has to be greater than 0!")
                continue
            }
            stake(amount)
        }
    }
}
```

5. In case the user creates a new transaction, it is added to the nodes **mempool** and is also flooded across the network so that other nodes can add it to their mempool as well. The mempool represents unconfirmed transactions that a node is either part of or has heard about from its peers. The idea is that if any node is chosen as the miner for a block, it will include the transactions in its mempool into the newblock.



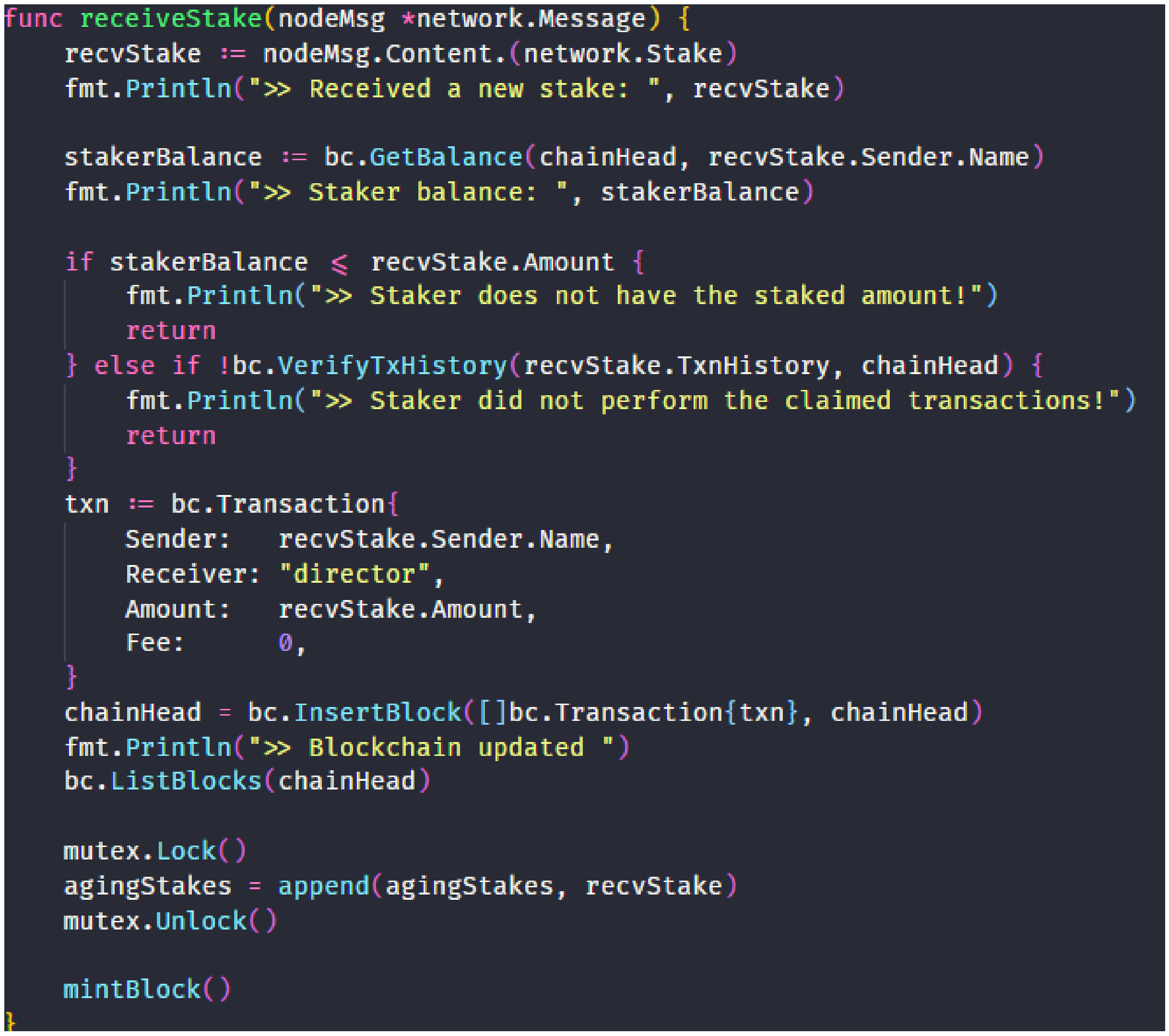
```go
func receiveTransaction(nodeMsg* network.Message) {
    recvTrans := nodeMsg.Content.(bc.Transaction)
    fmt.Println(">> Received transaction of", recvTrans.Sender)
    fmt.Println(">> Transaction: ", recvTrans)

    for _, tx := range memPool {
        if tx == recvTrans {
            fmt.Println(">> Transaction already part of Mempool!")
            return
        }
    }
    memPool = append(memPool, recvTrans)
    printMempool()
    propogateTransactionToPeers(recvTrans)
}
```

6. The user can also stake some amount in a bid to get selected to mine a new block. Once the user creates a stake, it is flooded across the network and also sent to the director. The staked amount now becomes part of the blockchain and the user cannot spend it anywhere else.



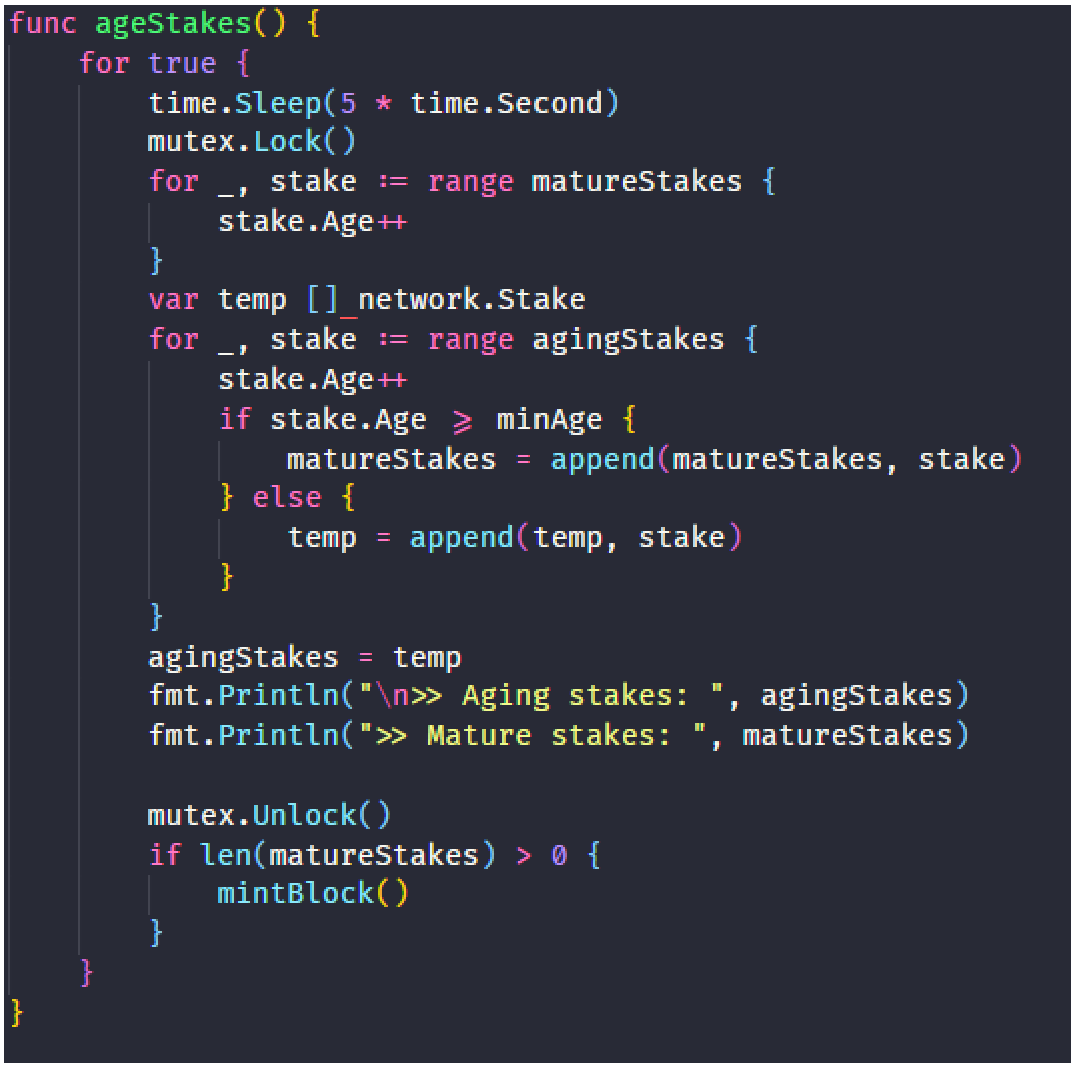
```go
func stake(stakeAmt float32) {
    myStake := network.Stake{
        Sender:         network.Peer{name, port},
        Amount:         stakeAmt,
        TxnHistory:     txnHistory,
        Age:            0,
        TimeOfCreation: time.Now(),
    }
    network.SendMessage(protocol.Receive_Stake, myStake, director)

    txn := bc.Transaction{
        Sender:   name,
        Receiver: "director",
        Amount:   stakeAmt,
        Fee:      0,
    }
    chainHead = bc.InsertBlock([]bc.Transaction{txn}, chainHead)
    fmt.Println(" >> Blockchain updated")
    bc.ListBlocks(chainHead)
    propogateStakeToPeers(myStake)
}
```

7. The director maintains two lists, one that represents new stakes and one that represents stakes that have matured. Our idea was that a stake has to sit for a certain amount of time before it can be considered again to prevent nodes with large stakes from continually mining new blocks. So when the director receives a new stake, it adds it to the **agingStakes** list.



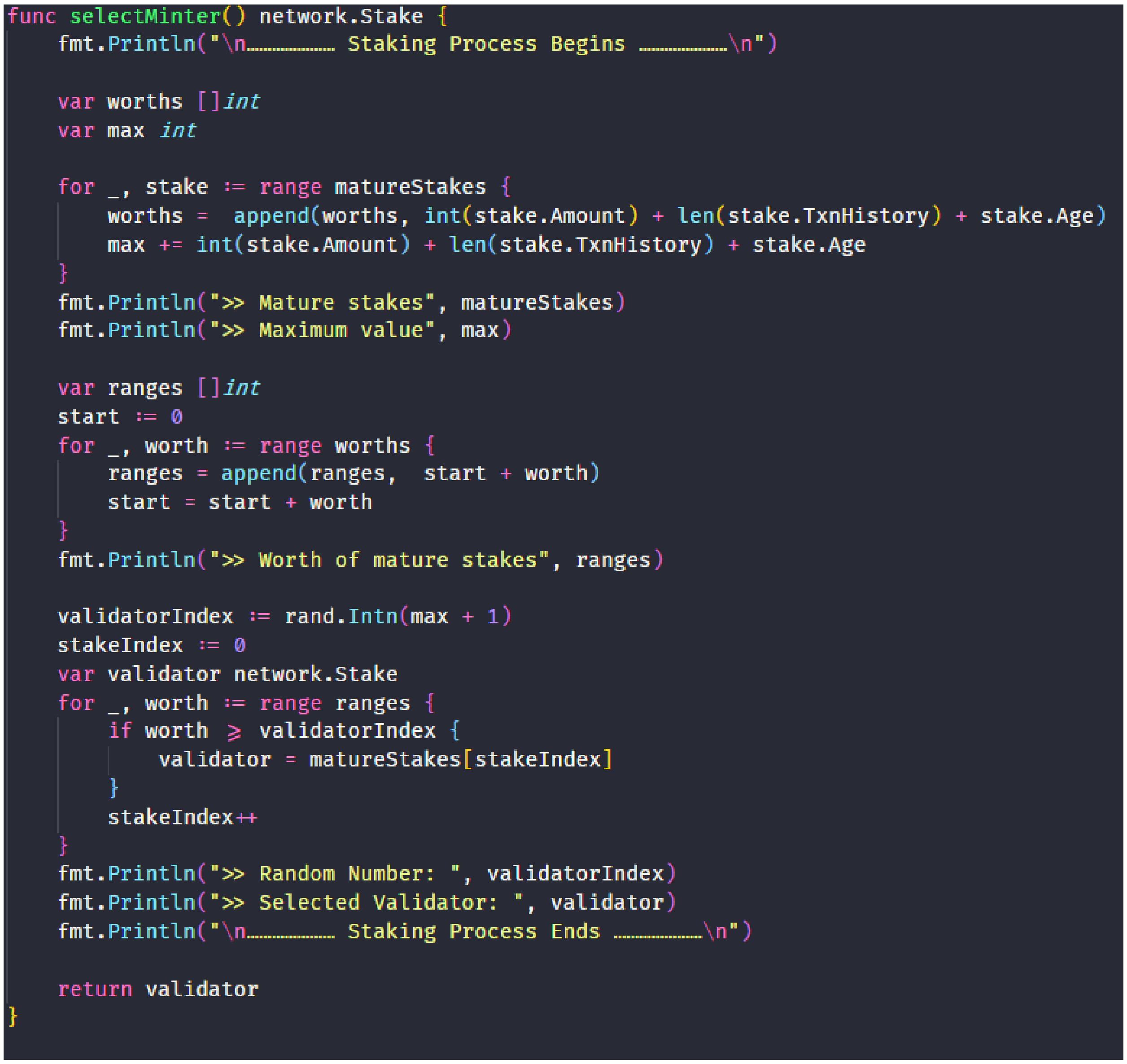
```go
func receiveStake(nodeMsg *network.Message) {
    recvStake := nodeMsg.Content.(network.Stake)
    fmt.Println(">> Received a new stake: ", recvStake)

    stakerBalance := bc.GetBalance(chainHead, recvStake.Sender.Name)
    fmt.Println(">> Staker balance: ", stakerBalance)

    if stakerBalance <= recvStake.Amount {
        fmt.Println(">> Staker does not have the staked amount!")
        return
    } else if !bc.VerifyTxHistory(recvStake.TxnHistory, chainHead) {
        fmt.Println(">> Staker did not perform the claimed transactions!")
        return
    }
    txn := bc.Transaction{
        Sender:   recvStake.Sender.Name,
        Receiver: "director",
        Amount:   recvStake.Amount,
        Fee:      0,
    }
    chainHead = bc.InsertBlock([]bc.Transaction{txn}, chainHead)
    fmt.Println(">> Blockchain updated ")
    bc.ListBlocks(chainHead)

    mutex.Lock()
    agingStakes = append(agingStakes, recvStake)
    mutex.Unlock()

    mintBlock()
}
```

8. The director also continually ages the stakes after a certain period of time and checks whether any stakes have reached maturity. If they have, they are transferred to the **matureStakes** list.



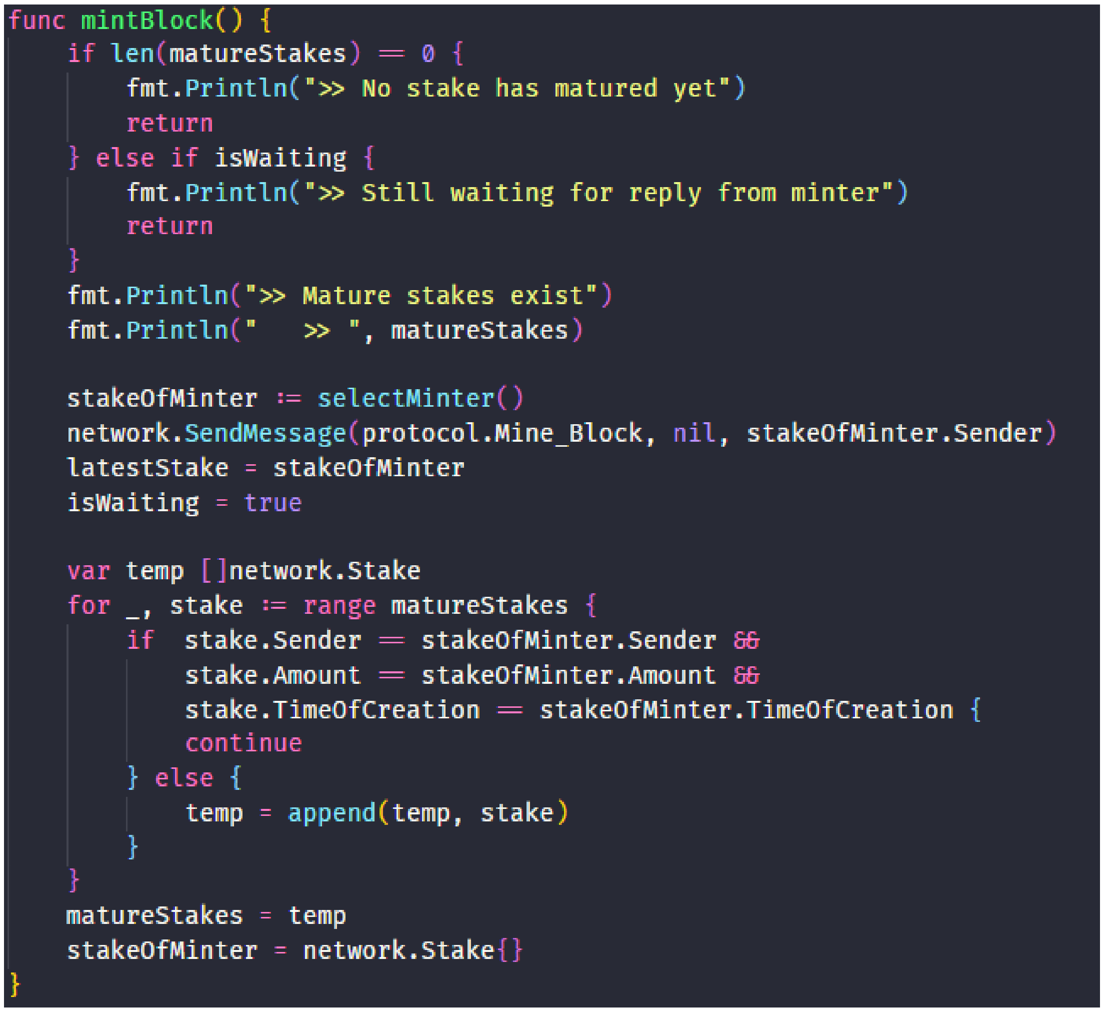
```go
func ageStakes() {
    for true {
        time.Sleep(5 * time.Second)
        mutex.Lock()
        for _, stake := range matureStakes {
            stake.Age++
        }
        var temp []_network.Stake
        for _, stake := range agingStakes {
            stake.Age++
            if stake.Age >= minAge {
                matureStakes = append(matureStakes, stake)
            } else {
                temp = append(temp, stake)
            }
        }
        agingStakes = temp
        fmt.Println("\n>> Aging stakes: ", agingStakes)
        fmt.Println(">> Mature stakes: ", matureStakes)

        mutex.Unlock()
        if len(matureStakes) > 0 {
            mintBlock()
        }
    }
}
```

9. If some stakes reach maturity, the director then begins mining a new block. This process begins with the selection of stake from those currently in the matureStakes list. The selection depends on the amount of time the stake has resided with the director, the amount of the stake and the total number of transactions made by the staking node, which is meant to gauge its level of activity. The method used to pick a winning stake is the fitness proportionate selection, described here.

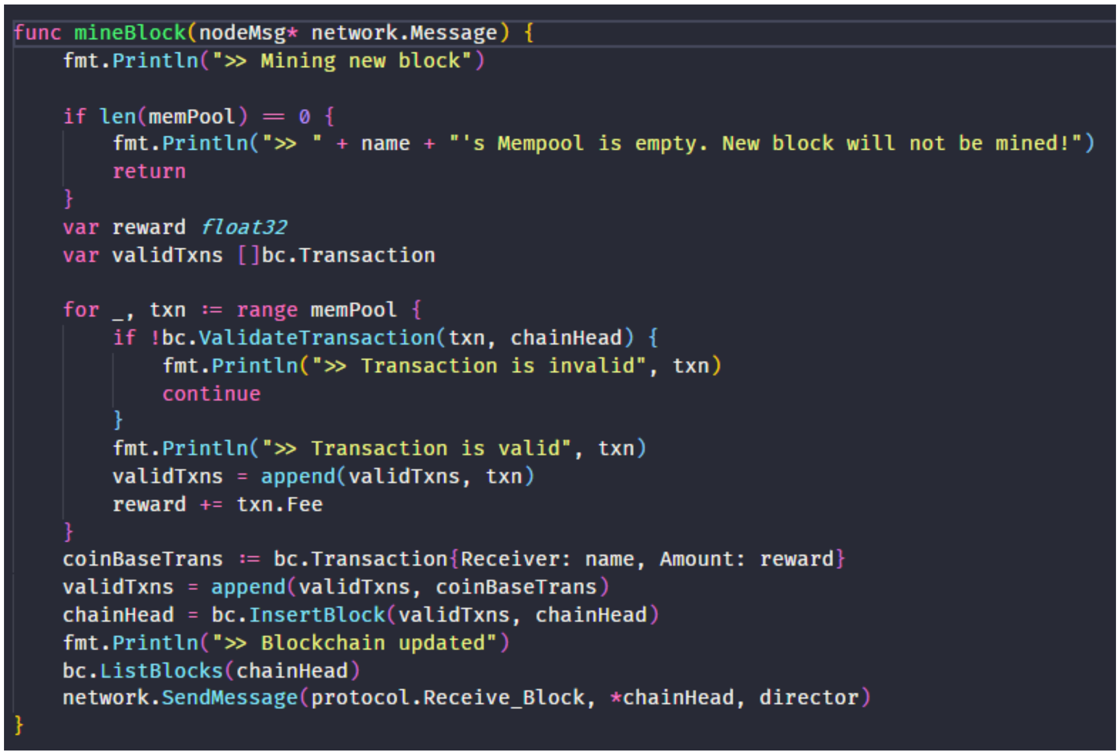


```go
func selectMinter() network.Stake {
    fmt.Println("\n.................... Staking Process Begins ....................\n")

    var worths []int
    var max int

    for _, stake := range matureStakes {
        worths =  append(worths, int(stake.Amount) + len(stake.TxnHistory) + stake.Age)
        max += int(stake.Amount) + len(stake.TxnHistory) + stake.Age
    }
    fmt.Println(">> Mature stakes", matureStakes)
    fmt.Println(">> Maximum value", max)

    var ranges []int
    start := 0
    for _, worth := range worths {
        ranges = append(ranges,  start + worth)
        start = start + worth
    }
    fmt.Println(">> Worth of mature stakes", ranges)

    validatorIndex := rand.Intn(max + 1)
    stakeIndex := 0
    var validator network.Stake
    for _, worth := range ranges {
        if worth >= validatorIndex {
            validator = matureStakes[stakeIndex]
        }
        stakeIndex++
    }
    fmt.Println(">> Random Number: ", validatorIndex)
    fmt.Println(">> Selected Validator: ", validator)
    fmt.Println("\n.................... Staking Process Ends ....................\n")

    return validator
}
```

10. The winner is asked by the director to propose a new block. Upon receiving this message, the winner proceeds to collect the transactions in its mempool and add them to its new block. It also adds a new transaction where it collects all the fees in the included transactions as it mining reward.

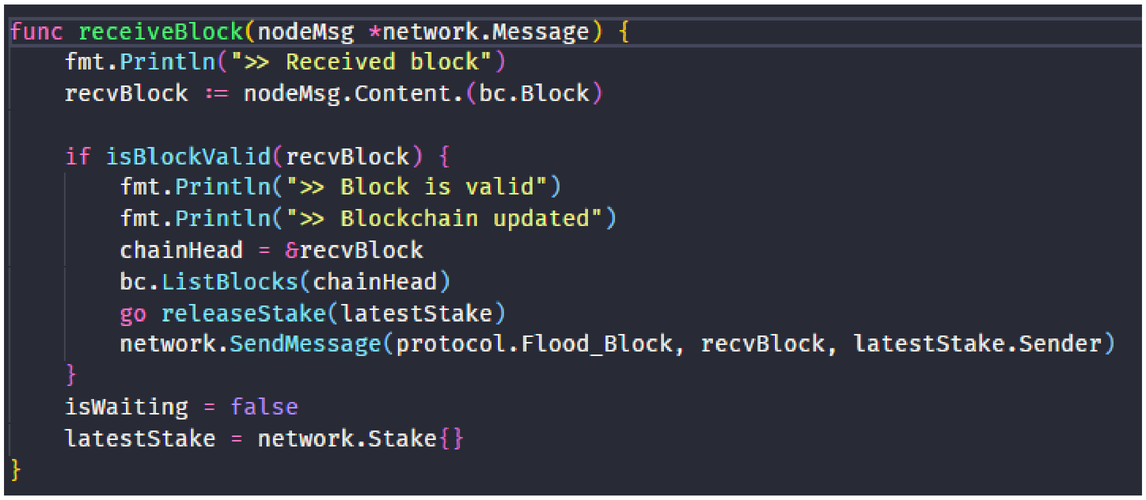


```go
func mintBlock() {
    if len(matureStakes) == 0 {
        fmt.Println(">> No stake has matured yet")
        return
    } else if isWaiting {
        fmt.Println(">> Still waiting for reply from minter")
        return
    }
    fmt.Println(">> Mature stakes exist")
    fmt.Println("    >> ", matureStakes)

    stakeOfMinter := selectMinter()
    network.SendMessage(protocol.Mine_Block, nil, stakeOfMinter.Sender)
    latestStake = stakeOfMinter
    isWaiting = true

    var temp []network.Stake
    for _, stake := range matureStakes {
        if  stake.Sender == stakeOfMinter.Sender &&
            stake.Amount == stakeOfMinter.Amount &&
            stake.TimeOfCreation == stakeOfMinter.TimeOfCreation {
            continue
        } else {
            temp = append(temp, stake)
        }
    }
    matureStakes = temp
    stakeOfMinter = network.Stake{}
}
```

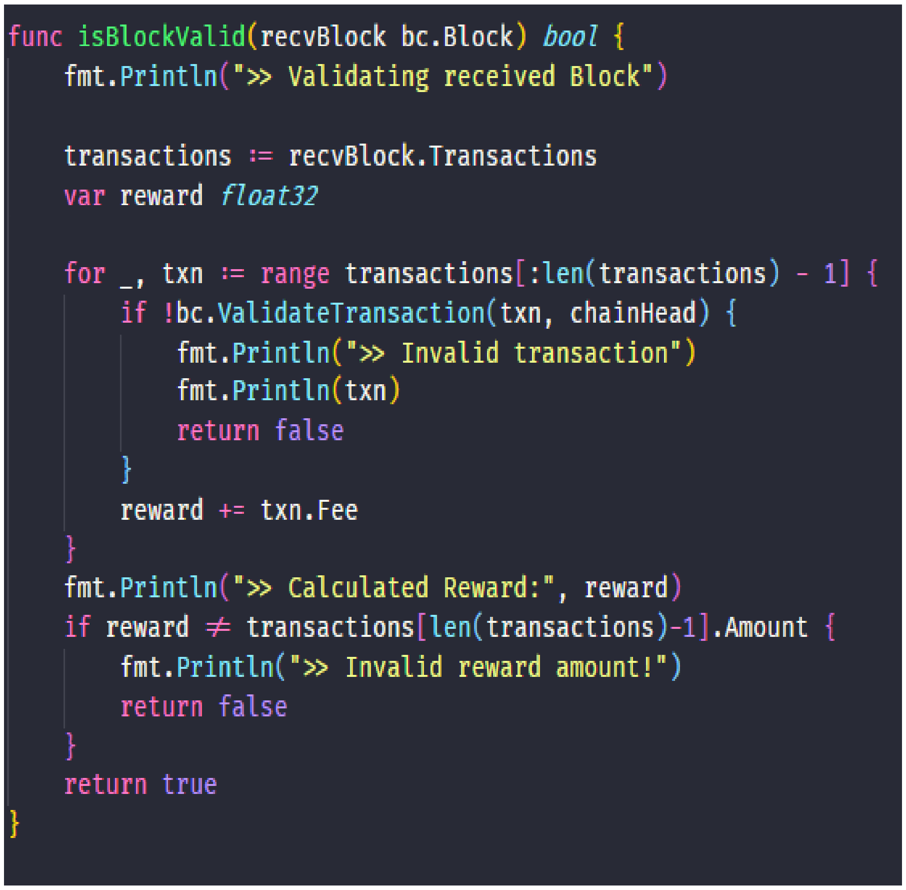


```go
func mineBlock(nodeMsg* network.Message) {
    fmt.Println(">> Mining new block")

    if len(memPool) == 0 {
        fmt.Println(">> " + name + "'s Mempool is empty. New block will not be mined!")
        return
    }
    var reward float32
    var validTxns []bc.Transaction

    for _, txn := range memPool {
        if !bc.ValidateTransaction(txn, chainHead) {
            fmt.Println(">> Transaction is invalid", txn)
            continue
        }
        fmt.Println(">> Transaction is valid", txn)
        validTxns = append(validTxns, txn)
        reward += txn.Fee
    }
    coinBaseTrans := bc.Transaction{Receiver: name, Amount: reward}
    validTxns = append(validTxns, coinBaseTrans)
    chainHead = bc.InsertBlock(validTxns, chainHead)
    fmt.Println(">> Blockchain updated")
    bc.ListBlocks(chainHead)
    network.SendMessage(protocol.Receive_Block, *chainHead, director)
}
```

11. The miner then sends this new block back to the director. The director now checks if the proposed block is valid by checking all the included transactions as well the reward claimed by the miner.

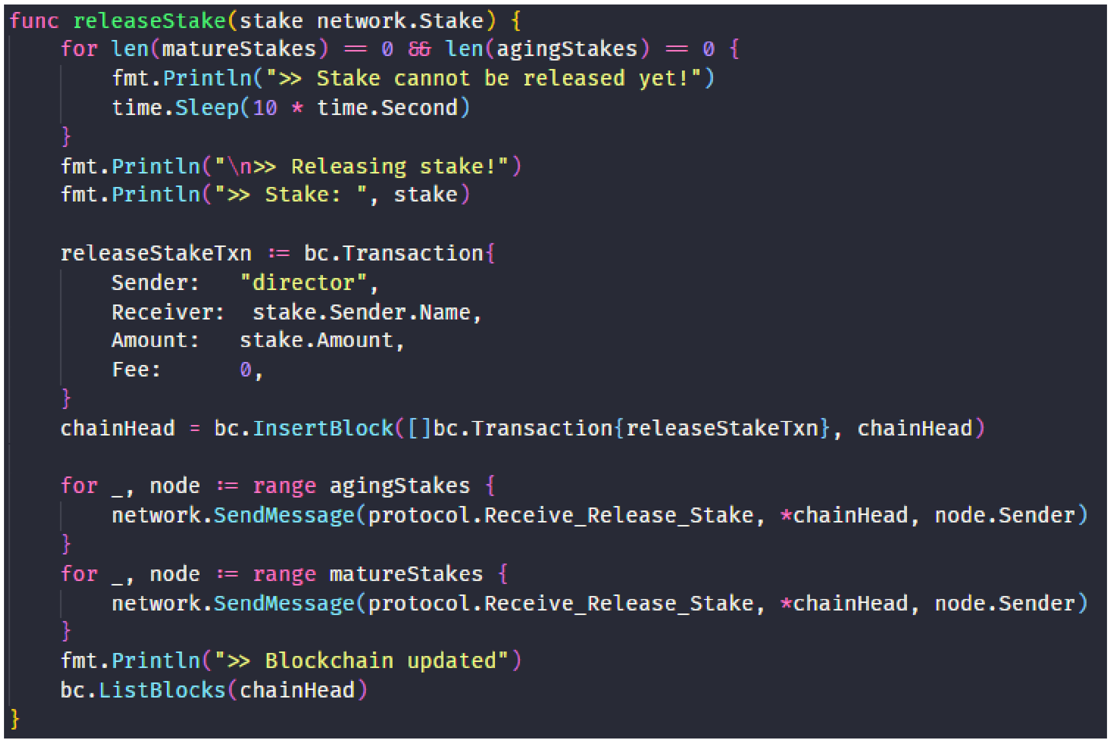


```go
func receiveBlock(nodeMsg *network.Message) {
    fmt.Println(">> Received block")
    recvBlock := nodeMsg.Content.(bc.Block)

    if isBlockValid(recvBlock) {
        fmt.Println(">> Block is valid")
        fmt.Println(">> Blockchain updated")
        chainHead = &recvBlock
        bc.ListBlocks(chainHead)
        go releaseStake(latestStake)
        network.SendMessage(protocol.Flood_Block, recvBlock, latestStake.Sender)
    }
    isWaiting = false
    latestStake = network.Stake{}
}
```

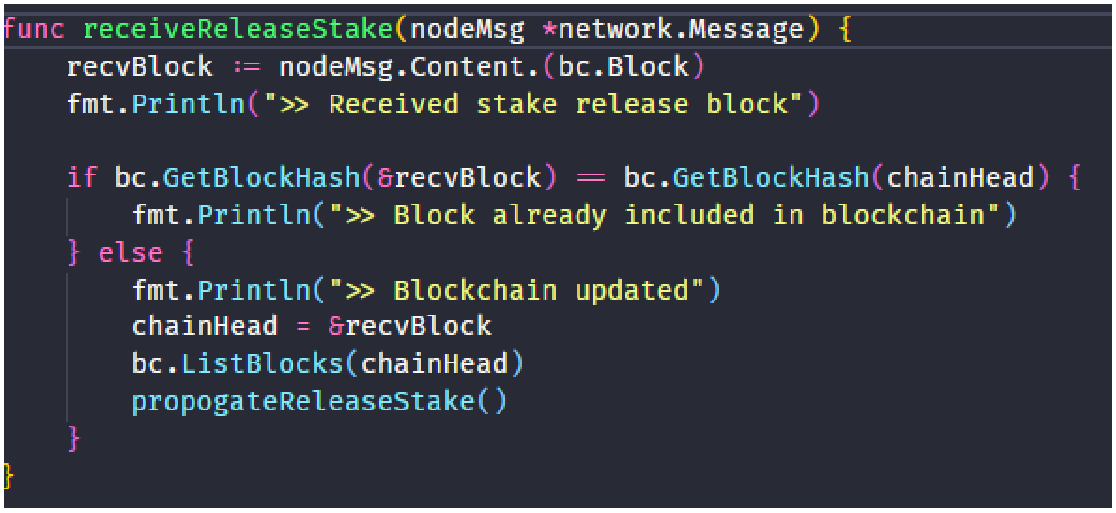


```go
func isBlockValid(recvBlock bc.Block) bool {
    fmt.Println(">> Validating received Block")

    transactions := recvBlock.Transactions
    var reward float32

    for _, txn := range transactions[:len(transactions) - 1] {
        if !bc.ValidateTransaction(txn, chainHead) {
            fmt.Println(">> Invalid transaction")
            fmt.Println(txn)
            return false
        }
        reward += txn.Fee
    }
    fmt.Println(">> Calculated Reward:", reward)
    if reward ≠ transactions[len(transactions)-1].Amount {
        fmt.Println(">> Invalid reward amount!")
        return false
    }
    return true
}
```

12. If the miner tries to cheat in any way, then its proposed stake will remain in the blockchain and the staked amount will be lost. If, however the proposed block check outs, the director will release the stake by creating a special transaction crediting the miner with the original staked amount. This transaction will be added to the blockchain and flooded across the network through the nodes whose stakes are in waiting to be considered.



```go
func releaseStake(stake network.Stake) {
    for len(matureStakes) == 0 && len(agingStakes) == 0 {
        fmt.Println(">> Stake cannot be released yet!")
        time.Sleep(10 * time.Second)
    }
    fmt.Println("\n>> Releasing stake!")
    fmt.Println(">> Stake: ", stake)

    releaseStakeTxn := bc.Transaction{
        Sender:   "director",
        Receiver:  stake.Sender.Name,
        Amount:   stake.Amount,
        Fee:      0,
    }
    chainHead = bc.InsertBlock([]bc.Transaction{releaseStakeTxn}, chainHead)

    for _, node := range agingStakes {
        network.SendMessage(protocol.Receive_Release_Stake, *chainHead, node.Sender)
    }
    for _, node := range matureStakes {
        network.SendMessage(protocol.Receive_Release_Stake, *chainHead, node.Sender)
    }
    fmt.Println(">> Blockchain updated")
    bc.ListBlocks(chainHead)
}
```



```go
func receiveReleaseStake(nodeMsg *network.Message) {
    recvBlock := nodeMsg.Content.(bc.Block)
    fmt.Println(">> Received stake release block")

    if bc.GetBlockHash(&recvBlock) == bc.GetBlockHash(chainHead) {
        fmt.Println(">> Block already included in blockchain")
    } else {
        fmt.Println(">> Blockchain updated")
        chainHead = &recvBlock
        bc.ListBlocks(chainHead)
        propogateReleaseStake()
    }
}
```