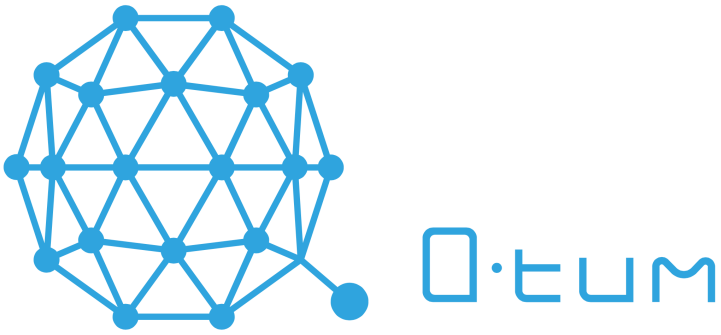
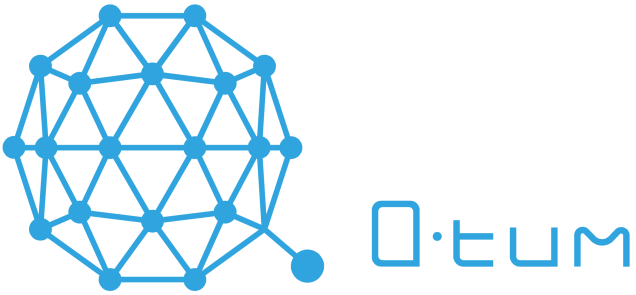
Qtum 部分设计和实现过程

（完整设计和实现文档预计在 6 月份发布）2017/5/29



说明：本文档只是部分 JIRA task 的汇总和摘要，只包含了 Qtum 的一部分设计过程，并且随着 Qtum 项目开发的快速进展，有一些内容会和最终的测试网络不一致，本文档的目的只在于帮助大家了解 Qtum 的开发过程，从中可以了解开发团队在过去一年中做的部分工作。本文档只包含了 Qtum Core 的一些简单实现工作，并不包含 Qtum 设计过程，钱包、浏览器、移动端设计、测试、研究等工作，以上提到的其他工作，Qtum 开发团队也基本上已经完成。测试网络预计 6 月份发布。 **（备注：目前 Qtum 团队合计 25 人，核心开发者 20 人，如果有兴趣加入 Qtum，请发简历到 job@qtum.org ）**

Qtum 整个项目的实现过程大概花费了 1.5 年的时间，原型想法开始于 2016 年 3 月份，在 2016 年 3 月，领导技术团队搭建基于以太坊的企业级应用的时候，发现了很多局限性，并且结合自己读博士期间的研究和过去 5 年在区块链行业经历的一切和思考，以及对区块链行业未来的判断，（优酷视频：2 个小时解读 量子链的过去 现在 和未来），当时考虑的是如何为现有的生态增加价值，并找到 Qtum 的独特定位，能够在中国搭建一个优质的区块链技术社区。



比特币带来的技术革命的开端，以太坊把智能合约从概念变成了现实，如何站在现有的生态上走出一条激励相容的道路是一个挑战，答案也只有一个：那就是创新。

Qtum 在比特币 UTXO 模型的基础上，做了账户的抽象，从而可以支持多种虚拟机（EVM 是其中一个，Lua 和 Bloq-ora 也在实现过程中），并且解耦了普通交易和合约交易，底层的共识机制采用拓展性更好的 POS 机制，并且创造了分布式自治协议（DGP）， 使基于机器节点共识的决策共识可以民主化和分布式。

我们相信 Qtum 的实现为现有的区块链生态带来了价值，Qtum AAL 打通了 UTXO 模型和虚拟机之前的鸿沟，Qtum DGP 使区块链的分布式决策和升级成为可能，即将公布的 Qtum MPOS 极大的提高了传统 POS 机制的安全性。更多虚拟机的支持和闪电网络以及侧链的开发，以及后期无限扩展的共识机制等也在我们的开发日程中，Qtum 会有快速的技术迭代。

另外随着 Qtum 测试网络的发布，我们相信我们也培养了中国区块链领域最好的一支技术团队，他们之前大多就职于百度 腾讯 阿里，毕业于北大 北邮 中科院，现在他们精通比特币和精通以太坊，并且全程参与了 Qtum 的设计和开发，这也是 Qtum 项目的另外一个目的吧，希望在中国，有更多的人参与到区块链的技术社区，推动技术的发展。

如果说我们这一代人还有什么优势的话，那就是这一代年轻人洞悉中国移动互联网带来的翻天覆地的变化，我们知道如何把区块链的易用性做到最好，这一切也将助力 Qtum 在中国和世界的发展。

未来已来，只是还没有开始流行。

QTUMCORE-1

Basic Qtum Branding

Do some basic renaming of the Bitcoin code to make it into Qtum. This shouldn't be too extensive, but should cover the following:

1. Use ~/.qtum instead of ~/.bitcoin for data directory by default
2. Build system should output qtumd instead of bitcoind/cli etc

QTUMCORE-2

Create basic Qtum network

Change the parameters so that the Qtum network is different from Bitcoin.

1. Change genesis seed
2. Change P2P/RPC ports
3. Change "magic" P2P network bytes.
4. Add whatever changes are needed so that a new mainnet network can be mined and synced.
5. Remove bitcoin checkpoints and node lists
6. Make sure that all 3 networks work: mainnet, testnet, and regtest. With mainnet and testnet being distinctly different and incompatible with Bitcoin

Make sure to remove irrelevant tests that only affect Bitcoin, and to change tests which rely on network parameters

specific params:

Block time: 128 seconds
Block size: 2Mb
P2P port: 3888 (chosen arbitrarily)
RPC port: 3889
testnet P2P port: 13888
testnet RPC port: 13889
Qtum pubkeyhash address: 58 (Q)
Qtum p2sh address: 50 (M)
Qtum testnet p2pkh: 120 (q)
Qtum testnet p2sh: 110 (m)

QTUMCORE-3

Add EVM and OP_CREATE for contract execution

After this story, the EVM should be integrated and a very basic contract should be capable of being executed.

There will be a new opcode, OP_CREATE (formerly OP_EXEC), which takes 4 arguments, in push order:

1. VM version (currently 1 is EVM)
2. Gas price (not yet used, anything is valid)
3. Gas limit (not yet used, assume very high limit)
4. bytecode

For now it is OK that this script format be forced and mandatory for OP_CREATE transactions on the blockchain. (ie, only "standard" allowed on the blockchain)

When OP_CREATE is encountered, it should execute the EVM and persist the contract to a database (triedb)

Note:
Make sure to follow policy for external code (commit vanilla unmodified code first, and then change it as needed)
Make the EVM test suite functional as well (someone else can setup continuous integration changes for it though)

QTUM Address for new contracts
QTUNCORE-3 Story. Test that qtum address is generated after contract creation. Before running the test make sure that:

- Node started в –regtest
- Bitcoin tests passed

Test Steps
1. Create transaction TX_DEPLOYMENT in bitcoinlib-js, run decoderawtransaction.
2. Create transaction TX_ASSIGN_SC in bitcoinlib-js, run decoderawtransaction.
3. Create transaction TX_DEPLOYMENT in bitcoinlib-js, run sendrawtransaction.
3. Create transaction TX_ASSIGN_SC in bitcoinlib-js, run sendrawtransaction.
Expected Result
1. Transaction marked as deployment.
2. Transaction marked as assign.
3. Contract created (address and balance are displayed on the console).
4. Contract state changed (address and balance are displayed on the console).

QTUMCORE-4

Add PoS block verification and mining

After this story the Qtum blockchain should be capable of verifying basic PoS v3 blocks, and the wallet should be able to mine these blocks similar to PoW blocks. There is not a network rule as of yet for only allowing PoS blocks at this point, so the chain will accept either PoW and PoS blocks as valid. Both block types should use the same default bitcoin standard difficulty.

Thing to ensure:

stake modifier changes every block
Coinage does not change PoS values
No decentralized checkpoints (still in design)

Make sure that these statements remain true:

- don't enforce tx fee as consensus rule
- static pos reward (make it just 1 token for now, will change later)
- stake modifier v2 - The purpose of stake modifier is to prevent a txout (coin) owner from computing future proof-of-stake generated by txout at the time of transaction confirmation. Verification of the block was changed, now you need only previous block, so all other blocks can be deleted and space saved (you need last 500 blocks to stake) pruning capabilities should be in future major release
- compatibility with BIP66
- only last 500 blocks are needed to verify proof of stake
- No coin age, instead just a coin confirmation requirement (500 blocks is fine for now)
-

- ## QTUMCORE-7

Add "createcontract" RPC call

A new RPC call should be added call "" "createcontract". This RPC call will be used to deploy a new smart contract to the Qtum blockchain.

Syntax:

createcontract gas-price gas-limit bytecode [sender-address] [txfee] [broadcast]

sender-address is optional. If no address is specified, then it should be picked randomly from the wallet. If no outputs exist to spend using that sender-address, then an error should be shown and no transaction created.

txfee is optional and if not specified should use the same auto txfee as the rest of the wallet (for example sendtoaddress uses an auto txfee)

broadcast should default to true. If broadcast is false, then the transaction is created and signed, and then printed to the screen in hex rather than broadcast to the network.

If the sender-address does have an output, but it is not enough to cover the gas costs and tx fees, then any UTXO owned by the wallet should be used by the transaction to cover those fees. (not all funds must come from sender-address, but the sender-address must be vin[0])

After execution, if broadcast is true, it should print the txid and the new contract address.

QTUMCORE-10

Add ability for contracts to call other deployed contracts

Contracts should be capable of calling other contracts using a new opcode, OP_CALL.
Arguments in push order:
version (32 bit integer)
gas price (64 bit integer)
gas limit (64 bit integer)
contract address (160 bits)
data (any length)
OP_CALL should ways return false for now.
OP_CALL only results in contract execution when used in a vout; Similar to OP_CREATE, it uses the special rule to process the script during vout processing (rather than when spent as is normal in Bitcoin). Contract execution should only be triggered when the transaction script is in this standard format and has no extra opcodes.
If OP_CALL is created that uses an invalid contract address, then no contract execution should take place. The transaction should still be valid in the blockchain however. If money was sent with OP_CALL, then that money (minus the gas fees) should result in a refund transaction to send the funds back to vin[0]'s vout script.
The "sender" exposed to EVM should be the pubkeyhash spent by vin[0]. If the vout spent by vin[0] is not a pubkeyhash, then the sender should be 0.
Funds can be sent to the contract using an OP_CALL vout. These funds will be handled by the account abstraciton layer in a different story, to expose this to the EVM.
Multiple OP_CALLS can be used in a single transaction. However, before contract execution, the gas price and gas limit of each OP_CALL vout should be checked to ensure that the transaction provides enough transaction fees to cover the gas. Additionally, this should be verified even when the contract is not executed, such as when it is accepted in the mempool.

QTUMCORE-13

Add "sendtocontract" RPC call

An rpc call should be adding for sending data and (optionally) money to a contract that has been deployed on the blockchain.

The format should be:

sendtocontract contract-address data (value gaslimit gasprice sender broadcast)

This should create a contract call transaction using OP_CALL.

Value defaults to 0.
sender-address is optional. If no address is specified, then it should be picked randomly from the wallet. If no outputs exist to spend using that sender-address, then an error should be shown and no transaction created.
broadcast should default to true. If broadcast is false, then the transaction is created and signed, and then printed to the screen in hex rather than broadcast to the network.
If the sender-address does have an output, but it is not enough to cover the gas costs, tx fees, and value, then any UTXO owned by the wallet should be used by the transaction to cover the remainder. (not all funds must come from sender-address, but the sender-address must be vin[0])
After execution, if broadcast is true, it should print the txid and the new contract address.

QTUMCORE-14

Add "callcontract" RPC call for off-chain computations

There should be an RPC call that executes a contract without requiring interaction with the blockchain network, and thus without gas or other fees.

callcontract contract-address data (sender)

This should execute the contract locally, and if the contract function returns data, it should be returned/printed by the RPC call. Sender is optional and does not require an owned vout (it can be any valid address)

QTUMCORE-16

Change mainnet to only allow PoS after a certain blockheight

QTUMCORE-18

Put EVM "LOG" opcode outputs into an accessible log file

There should be a file that holds all of the data from EVM LOG opcodes, so that it can be used for testing and debugging. There are ITDs written for this

The log file should only be used when `-record-log-opcodes` is passed into qtumd

ITD: https://github.com/qtumproject/qtum-itds/blob/master/evm/log%20opcode%20logging.md

QTUMCORE-22

Setup continuous integration

Setup a Jenkins server and make it so that it does:

1. Automatically builds qtum on all branches on Linux and reports build errors
2. Automatically runs the Qtum (Bitcoin) test suite, reporting any failing tests
3. Tears down any VMs constructed to ensure we don't use more VM time than necessary

QTUMCORE-23

Change all output strings Bitcoin -> Qtum, BTC -> QTUM

Replace all instances of bitcoin and BTC with qtum and QTUM (Just the output strings, not the actual class/variable names and not the code comments)

QTUMCORE-26

Increase Script limits to suitable values

There are a number of limits in vanilla Bitcoin that restrict our usage of smart contracts. These limits should be relaxed:

- maximum data push size: 1Mb
- maximum stack size: 1Mb
- Any other limits that we run into with large scripts caused by contracts?

QTUMCORE-27

Consensus/chain parameters for PoS

The consensus provide parameters that will affect the creation of new blocks in the network.

The PoS limit need to be added in order to determine the initial difficulty for PoS.

The halving interval is 4 years and there will be 7 halving intervals.

The last PoW block need to be defined and set to 5000 for the main net. The reward for PoW block is 20000 coins, the reward for PoS block is 4 coins.

QTUMCORE-28

Update of block/block-header parameters for PoS

The general rule for the header is to keep it as small as possible.

Four parameters are needed for PoS in order to work:

- Block signature, that is the signature of the whole block, signed by the block creator
- Block type (Proof Of Work or Proof Of Stake)
- Previous stake location, need in order to confirm the block validity, mandatory for PoS
- Staking time, the time when the staking transaction was created, mandatory for PoS

Block signature must be a parameter. The parameter need to be in the header due to the segregated witness which split the block delivery into header and transactions that can be required from any user.

The other parameters can be extracted from the PoS transaction. Whether or not to include those 3 parameters in the header too can be discussed. The block header is downloaded before the whole block, not including those parameters in the header will prevent doing PoS checks to the header before downloading the whole block. Blackcoin download the whole block before the checks and put those parameters into the Block.

QTUMCORE-29

Proof of Work/Stake kernel update

The source code for PoW and PoS kernel is kept into the files:

- pos.h
- pos.cpp
- pow.h
- pow.cpp

The location of the files is according to master-pos, in Blackcoin the location is kernel.h and kernel.cpp.

We need to decide if we will keep it or use different structure for organization of the code for PoW and PoS kernel.

The version of the stake modifier that will be used is the second version.

Block difficulty, masks, modifiers, time-stamps are all used hare, so this is important parts for PoS.

QTUMCORE-30

Miners for Quantum

The miners are tools for pushing blocks to the network automatically.
We need them to avoid using generate to push new blocks and for the network being up.

The PoW mining have a lot of options due to the use of Bitcoin miners.

- Internal Miner
- CPU miners
- Graphic card miners

PoW mining will be used only for the initial blocks for establishing the stake holders.
So any miner will be sufficient for PoW.

PoS miner is needed after the mining of the initial PoS blocks for staking.
The PoS miner is the one used in Blackcoin.

Parameters are needed for activating and deactivating the miners when Quantum is started.

The parameter "gen" was used to start the "Internal Miner" for PoW, the parameter "staking" to start the staking miner. For Quantum that might be different ...

QTUMCORE-31

Update the maximum amount of coins

File amount.h contain the code for the maximum coins:

```
static const CAmount MAX_MONEY = 21000000 * COIN;
```

PoS doesn't have halving interval, so the sum will diverge (infinity).

The MAX_MONEY need to be updated to the maximum supported number.

QTUMCORE-32

Add coinbase and coinstake checks in Transaction

The check will determine the type of transaction.

Coinbase check if the input coins have no previous output that generate them. Those are the new coins created as reward for mining PoW block.

Coinstake check if the input coins have previous output that generate them and the first output of the transaction is empty, the second or the third is the reward. The first output is empty to prevent it being mistaken for normal transaction. If the input is small then the output coins are contained in the second output, otherwise they are splitted between the second and the third output.

We can add additional check called IsNormal (or other more suitable name) that will be used to define normal transaction, not a generated one like the PoW and PoS transactions.

Also insure that no OP_CREATE nor OP_CALL transactions can be used for staking

QTUMCORE-33

Add time-stamp in Transaction for PoS

The time-stamp is the time that the transaction has been created. It is used for PoS to avoid including transactions in the block that have time bigger then the block time. Transaction with bigger time will eventually be include in the chain when the value for the time become less then the one from the block time.

The other uses for the parameter is to determine the staking time for the block (the time of PoS transaction).

QTUMCORE-34

Signing PoS block

The creator of new PoS block need to sign the block so the other miners can verify that the block is created from the miner and check its validity.

According to PoS 3, the first transaction is empty in case of PoS. The second transaction is the PoS transaction in the block.

The block signature is empty in case of PoW - the first transaction in the block is not empty for PoW.

The block version should be bumped to 2 for all PoS blocks

QTUMCORE-35

Creating new PoS block

The first transaction in PoS block is empty.
The second transaction in PoS block contain the PoS transaction.

The Coins are considered as available if the transaction outputs that create them are matured.
List of available coins for staking need to be created, UTXO from the list will be chosen and used to create the staking transaction.

The staking coins are used to create the PoS transaction. It is special type of transaction that the output is bigger then the input, that mean if the input coins were 1000, the output coins will be 1001 coins due to the reward. Normal transaction have outputs that are smaller or equal to the inputs, PoS transaction is an exception from that rule.

If the stake is big, for PoS big stake is considered more then 200 coins, then the output will contain two entries, for example: Input is 1000, outputs are 500.5 and 500.5. This parameter should be easily adjusted later (as it might need to be)

After creating the PoS transaction, the block is filled with transactions and is signed.

QTUMCORE-36

Accepting blocks for PoS

The block header need to be checked when received.

The whole block need to be checked before accepting it (when the block was created by the miner or received from other chains ...).

When processing PoS blocks we need to handle duplicate hashes and orphan blocks.

QTUMCORE-37

Coins serialization and transaction undo for PoS

The coins are saved when a transaction is considered as accepted (the block it pushed to the main chain) so they will not be able to be double spent.

The formula for saving coins need to be updated in order to provide flag to the coins that belong to CoinStake transaction. They will not be spendable until they mature.

The top of the chain might change if someone create a block that have more work. So in that case undo is needed to the transactions of the previous block. Update of the formula will be needed to include the CoinStake transactions too in that process.

QTUMCORE-38

RPC update for PoS information

Information about the PoS parameters need to be provide like the Difficultly, the type of mining used in the Block (Proof Of Work or Proof Of Stake), the reward for the PoS transaction.

QTUMCORE-39

Show staking information to the user in the Wallet

Update of the wallet if needed to get information for the staking and display them in a label in the GUI.

The wallet need to be updated so will correctly display the available coins for spending not considering the allocated coins in the staking process.

QTUMCORE-40

Block verification update

When the Wallet/Daemon is started the blocks are verified for errors against 4 levels of verification.

The default level is 3, but the code need to pass all 4 levels in order to be considered valid.
Update is needed when creating the Block Index to read the PoS parameters from the block database.

QTUMCORE-41

List of TODOs before testnet launch

This is a list of things that will need to be changed before testnet launch.
These values were kept at values suitable for development:

1- Restore block maturity to 500
2- set BIPs nStartTime and nTimeout to actual values after the chain launches, or activate thme by default
3- revert DEFAULT_MAX_TIP_AGE to 24 * 60 * 60; before testnet release

QTUMCORE-42

Add indexing per address and time of block

Could be useful to add indexing per address and time of the block. To be based on https://github.com/bitpay/bitcoin

In the future, this could be used for insight explorer node deployment.

This task is not touching any other that is currently Open so can be merged separately

QTUMCORE-43

Add Reserve Balance

The reserve balance is the coins which are reserved and not used for staking. The parameter `nReserveBalance` define the reserve balance. It can be checked by searching in `Blackcoin` for `reservebalance`.
The following functionalities contain reserve balance and require update:

- `-reservebalance` parameter that can be set when the application is initialized or in the configuration file
- RPC call for the reserve balance
- reduce the stake and the available coins for staking by the reserve balance
- label for displaying the reserve balance into the `optionsdialog.ui` and update of the model (Qt Wallet)

- **QTUMCORE-44**

Add hashStateRoot to CBlockHeader

We need to have hashStateRoot value available in CBlockHeader in order to be able to do next actions:

- Rollback database state (DisconnectBlock, incorrect block, etc.)
- Run State initialization with some data (currently it's empty).

- **QTUMCORE-45**

-

Initialization of EVM environment (EnvInfo)

- EVM environment is required to get information about blocks from Solidity. This information can be operated by the contract.
-

QTUMCORE-46

Cover with tests QtumTxConverter and ByteCodeExec

Create unit tests for QtumTxConverter and ByteCodeExec

QTUMCORE-47

Add Account Abstraction Layer for contracts

The account abstraction layer is a way for EVM contracts to work on the UTXO based blockchain. This story should cover nearly the entire module.

Funds can be sent to a contract using OP_CALL. When a contract receives or sends funds, it should result in a "condensing transaction". This condensing transaction will

spend any existing contract vouts which require their balance to be changed, and the outputs will be the new balance of the contracts.

The condensing transaction is created so that a contract never owns more than 1 UTXO. This significantly simplifies coin picking, and prevents many attack vectors with filling up a block.

There can be more than 1 condensing transaction per block. In the case of a single contract having multiple balance changes in a block, a condensing transaction might spend a previous condensing transaction's outputs in the same block. This is slightly wasteful but reduces the complexity of logic required, and allows for easily adding more contract execution transactions without needing to rewrite any previous transactions.

An ITD for the full behavior with all known edge cases is here: https://github.com/qtumproject/qtum-itds/blob/master/aal/condensing-transaction.md

Condensing Transaction Behavior

In the initial prototype we had an unsolved DoS problem where an attacker could send a contract many tiny outputs, and then spend them all at once, potentially exceeding the block size limit.

The solution which solves this is named "condensing transactions". Basically after a contract has completed it's execution, a single UTXO should hold all of it's tokens.

Environment

Fairly involved setup process, but can be done on mainnet or regtest

Test Steps

1. Deploy each Sender contract
2. Use `setSenders` for each contract to set them up to know about each other
3. Mine a block (block B1)
4. Send 8 coins to Sender1, method `share`(Transaction T1)
5. Mine a block (block B2)
6. Send 2 coins to Sender1, method `keep` (Transaction T2)

7. Send 2 coins to Sender1, method `sendAll` (Transaction T3)
8. Mine a block (Block B3)
9. Send 2 coins to Sender2, method share. Send a very low gas limit that causes it to run out of gas (Transaction T4)
10. Call Sender2, method `withdrawAll` (Transaction T5, sender address is A1)
11. Mine a block (Block B4)

Expected Results

At step 3, all 3 contracts should have a 0 balance and no UTXOs associated with them other than their creation transaction.

Block B2

At step 5, block B2 should contain 4 transactions:

1. Coinbase
2. Stake
3. T1
4. Condensing TX (Transaction C1)

T1 is of course unmodified.

C1 should contain 1 vin:

- Spend contract send (OP_CALL) in T1 using OP_TXHASH

And 3 vouts:

- C1.V0: version 0 OP_CALL to Sender1 of value 5 (8 initially, -4 for send to Sender2, then +1 for callback from Sender3)
- C1.V1: version 0 OP_CALL to Sender2 of value 2.5 (4 from Sender1, -2 for send to Sender3, then +0.5 for callback from Sender3)
- C1.V2: version 0 OP_CALL to Sender3 of value 0.5 (2 from Sender2, -1 for send to Sender1, -0.5 for send to Sender2)

If you check the balance of these contracts after this block, it should exactly match the outputs of C1. Furthermore, there should NEVER be more than 1 UTXO owned by a contract after contract execution is complete and the

condensing tx is complete. This could even be a network rule for safety against hidden attack vectors.

Now we do the sends for T2 and T3 and mine block C.

Block B3

Block B3 should contain 6 transactions:

1. Coinbase
2. Stake
3. T2
4. Condensing TX C2
5. T3
6. Condensing TX C3

C2 vins:

- T2 contract spend vout spent with OP_TXHASH
- C1.V0 --Sender1's balance

C2 vouts:

- C2.V0: version 0 OP_CALL to Sender1 of value 7 (5 from previous balance, +2 from T2)

C3 vins:

- T3 contract spend vout spent with OP_TXHASH
- C2.V0 -- Sender1's balance
- C1.V1 -- Sender2's balance

C3 vouts:

- C3.V0: version 0 OP_CALL to Sender2 of value 11.5 (2.5 from previous balance, +9 from Sender1's balance plus T3's contract coins)

Note that at this point Sender1 should now have no UTXOs owned by it at all, because it's balance is 0.

Block B4

Now block B4 is mined, it contains these transactions:

- Coinbase
- Stake
- T4
- Refund Transaction R1
- T5
- Condensing Transaction C4

R1 vins:

- T4 contract spend vout spent with OP_TXHASH

R1: vouts:

- R1.V0: pubkeyhash script (back to the first vin of T4) of value 2

C4 vins:

- C3.V0 -- Sender2's balance
- C1.V2 -- Sender3's balance (no coins sent with T5, so no input for it. We ignore 0-value outputs)

C4 vouts:

- C4.V0: pubkeyhash script for address A1, of value 12 (11.5 from Sender2 balance, +0.5 from withdrawal of Sender3's balance)

Now, at the end of our test, the end result is that all 3 of these contracts should now have a 0 balance and own no UTXOs.

Contract code

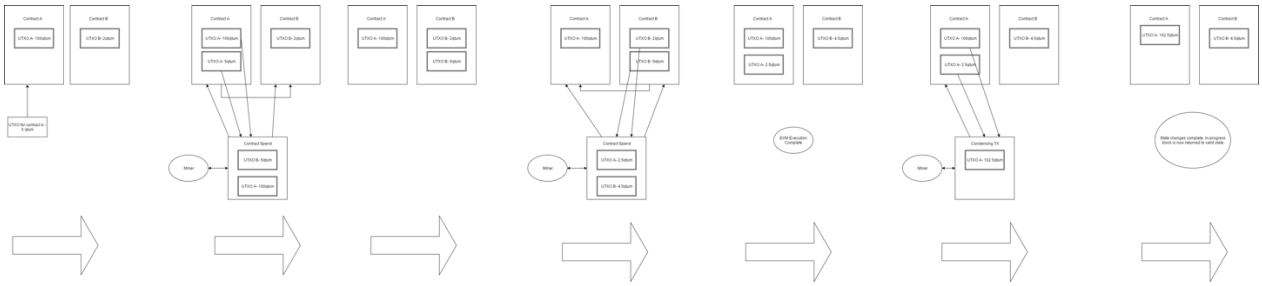
```
pragma solidity ^0.4.0;
contract Sender1 {
    Sender2 sender2;
    Sender3 sender3;
    function Sender1() {
    }
    function setSenders(address senderx, address sendery) public{
```

```
        sender2=Sender2(senderx);
        sender3=Sender3(sendery);
    }
    function share() public payable{
        if(msg.sender != address(sender3)){
            sender2.share.value(msg.value/2);
        }
    }
    function sendAll() public payable{
        sender2.keep.value(msg.value + this.balance);
    }
    function keep() public payable{
    }
    function() payable { } //always payable
}
contract Sender2{
    Sender1 sender1;
    Sender3 sender3;
    function Sender2() {
    }
    function setSenders(address senderx, address sendery) public{
        sender1=Sender1(senderx);
        sender3=Sender3(sendery);
    }
    function share() public payable{
        sender3.share.value(msg.value/2);
    }
    function keep() public payable{
    }
    function withdrawAll() public{
        sender3.withdraw();
        msg.sender.send(this.balance);
    }
    function() payable { } //always payable
}

contract Sender3 {
    Sender1 sender1;
    Sender2 sender2;
    function Sender3() {
    }
    function setSenders(address senderx, address sendery) public{
        sender1=Sender1(senderx);
        sender2=Sender2(sendery);
```

```
    }
    function share() public payable{
        sender1.share.value(msg.value/2);
        sender2.keep.value(msg.value/4);
    }
    function withdraw() public{
        msg.sender.send(this.balance);
    }
    function() payable { } //always payable
}
```

)



QTUMCORE-48

Improvements after PoC

Open for discussion after initial PoC reimplementation:

1. Remove Timestamp in tx
2. Base58 contract address
3. Reserve PoS balance unspendable
4. Discuss maturity blocks (1000?)
5. Contract Admin Interface

QTUMCORE-49

Research Attack Vectors after PoC

Attack Vectors:

1. Add minimum gas fee: We need to add minimum gas fee to avoid deploying 0 code contracts and pay no gas

- Maybe, but I think transaction fee might be ok to handle this
  2. Need to figure out how to deal with wei vs satoshi
- This is a hard one..
  3. Possible spam attack with EXEC ops not being considered as dust (using VM version 0 which does not pay gas)
- I think txfee should account for this, as well as version 0 potentially not being a standard transaction (still under consideration)
  4. Duplicate block spam - It might be possible to spam blocks with the same kernel hash but with different contents. Worst case being if they made it contain a lot of expensive transactions to validate, and then made the very last transaction invalid. This should trigger DoS conditions banning the node, but should be evaluated to see how expensive this actually is for nodes to handle
  5. Thoroughly consider the implications of segwit support being combined with PoSv3 and the security of SPV light wallets

- **QTUMCORE-51**

Formalize the version field for OP_CREATE and OP_CALL

In order to sustain future extensions to the protocol, we need to set some rules for how we will later upgrade and add new VMs by changing the "version" argument to OP_CREATE and OP_CALL.

We need a definitive VM version format beyond our current "just increment when doing upgrades". This would allow us to more easily plan upgrades and soft-forks. Proposed fields:

1. VM Format (can be increased from 0 to extend this format in the future): 2 bits
2. Root VM - The actual VM to use, such as EVM, Lua, JVM, etc: 6 bits
3. VM Version - The version of the Root VM to use (for upgrading the root VM with backwards compatibility) - 8 bits
4. Flag options - For flags to the VM execution and AAL: 16 bits

Total: 32 bits (4 bytes). Size is important since it will be in every EXEC transaction

Flag option bits that control contract creation: (only apply to OP_CREATE)

- 0 (reserve) Fixed gas schedule - if true, then this contract chooses to opt-out of allowing different gas schedules. Using OP_CALL with a gas schedule other than the one

specified in it's creation will result in an immediate exception and result in an out of gas refund condition

- 1 (reserve) Enable contract admin interface (reserve only, this will be implemented later. Will allow contracts to control for themselves what VM versions and such they allow, and allows the values to be changed over the lifecycle of the contract)
- 2 (reserve) Disallow version 0 funding - If true, this contract is not capable of receiving money through version 0 OP_CALL, other than as required for the account abstraction layer.
- bits 3-15 available for future extensions

Flag options that control contract calls: (only apply to OP_CALL)

- (none yet)

Flag options that control both contract calls and creation:

- (none yet)

These flags will be implemented in a later story

Note that the version field now MUST be a 4 byte push.

A standard EVM contract would now use the version number (in hex) "01 00 00 00"

Consensus behavior:

VM Format must be 0 to be valid in a block
Root VM can be any value. 1 is EVM, 0 is no-exec. All other values result in no-exec (allowed, but the no execution, for easier soft-forks later)
VM Version can be any value (soft-fork compatibility). If a new version is used than 0 (0 is initial release version), then it will execute using version 0 and ignore the value
Flag options can be any value (soft-fork compatibility). (inactive flag fields are ignored)

Standard mempool behavior:

VM Format must be 0
Root VM must be 0 or 1
VM Version must be 0
Flag options - all valid fields can be set. All fields that are not assigned must be set to 0

Defaults for EVM:

VM Format: 0
Root VM: 1
VM Version: 0
Flags: 0

QTUMCORE-52

Contract Admin Interface

(note, this isn't a goal for mainnet, though it would be a nice feature to include)

It should be possible to manage the lifecycle of a contract internally within the contract itself. Such variables and configuration values that might need to be changed over the course of a contract's lifecycle:

- Allowable gas schedules
- Allowable VM versions (ie, if a future VM version breaks this contract, don't allow it to be used, as well as deprecating past VM versions from being used to interact with this contract)
- Creation flags (the version flags in OP_CREATE)

All of these variables must be able to be controlled within the contract itself, using decentralized code. For instance, in a DAO scenario, it might be something that participants can vote on within the contract, and then the contract triggers the code that changes these parameters. In addition, a contract should be capable of detecting it's own settings throughout it's execution as well as when it is initially created.

I propose implementing this interface as a special pre-compiled contract. For a contract ot interact with it, it would call it using the Solidity ABI like any other contract.

Proposed ABI for the contract:

- `bytes[2048] GasSchedule(int n)`
- `int GasScheduleCount()`
- `int AddGasSchedule(bytes[2048]`
- `bytes[32] AllowedVMVersions()`
- `void SetAllowedVMVersions(bytes[32])`

Alternative implementations:
There could be a specific Solidity function which is called in order to validate that the contract should allow itself to be called in a particular manner:
```
pragma solidity ^0.4.0;
contract BlockHashTest {
function BlockHashTest() {
}

function ValidateGasSchedule(bytes32 addr) public returns (bool) {
if(addr=="123454")
{ return true; //allow contract to run }
```

```
return false; //do not allow contract to run
}
function ValidateVMVersion(byte version) public returns (bool){
if(version >= 2 && version < 10)
{ return true; //allow to run on versions 2-9. Say for example 1 had a vulnerability in it, and 10 broke the contract }

return false;
}
}
```

In this way a contract is responsible for managing it's own state. The basic way it would work is that when a you use OP_CALL to call a contract, it would first execute these two functions (and their execution would be included in gas costs). If either function returns false, then it immediately triggers an out of gas condition and cancels execution.

It's slightly complicated to manage the "ValidateVMVersion" callback however, because we must decide which VM version to use. A bad one could cause this function itself to misbehave.

QTUMCORE-53

Add opt-out flags to contracts for version 0 sends

Some contracts may wish to opt-out of certain features in Qtum that are not present in Ethereum. This way more Ethereum contracts can be ported to Qtum without worrying about new features in the Qtum blockchain

Two flag options should be added to the Version field, which only have an effect in OP_CREATE for creating the contract:

2. (1st bit) Disallow "version 0" OP_CALLs to this contract outside of the AAL. (DisallowVersion0)

- If this is enabled, then an OP_CALL using "root VM 0" (which causes no execution) is not allowed to be sent to this contract. If money is attempted to be sent to this contract using "version 0" OP_CALL, then it will result in an out of gas exception and the funds should be refunded. Version 0 payments made internally within the Account Abstraction Layer should not be affected by this flag.

Along with these new consensus rules, there should also be some standard mempool checks:

1. If an OP_CALL tx uses a different gas schedule than the contract creation, and the disallow dynamic gas flag is set, then the transaction should be rejected from the mempool as a non-standard transaction

(version 0 payments should not be allowed as standard transactions in the mempool anyway)

QTUMCORE-54

Add tests for Account Abstraction Layer for contracts

QTUMCORE-55

Decentralized Governance Protocol and Dynamic Gas Schedules

Decentralized Governance Protocol is a new concept to be implemented in Qtum. It allows us to change various network parameters without needing to release a new binary wallet. It's configuration is secured by a smart contract which implements multi-sig checks etc.

The first big concept to be controlled by DGP is to allow for gas schedules to be changed without needing to download a new binary wallet nor create a network fork. DGP is intended to control many things eventually, but to start with it will only be used for "dynamic gas schedules".

There are 2 primary contracts involved in each DGP feature:

1. The DGP framework contract for each feature
2. The actual DGP feature contract which controls the consensus data

Contract code: https://github.com/qtumproject/qtum-dgp/blob/master/dgp-template.sol.js

The DGP framework contract accomplishes the following:
1. Proposals and Voting - Each parameter change including internal key management is

proposed, and then voted on. If the vote meets chosen conditions, then the proposal is accepted and the operation is performed. A vote is counted using `msg.sender`, so that either a pubkeyhash address or a contract address can be used as a participant
2. Key management - It should be possible to add and remove participants, as well as changing parameters such as how many keys are required for a proposal to be accepted, for a key to be added, etc
3. Sending data of the right format to the DGP feature contract
4. Allow for itself to be disabled so that no further DGP changes can be made without a hard-fork to fix it (in case of a major exploit or problem)
5. Only one proposal is allowed at one time, a proposal can only be made by a participant. Each proposal should include an expiration of no more than 5,000 blocks. Every proposal expires after the expiration, or after voting is complete enough to reject and approve it.
6. (maybe) there should be a list of administrators which can remove proposals, and optionally be the only allowed parties to add a proposal?

Meanwhile, the DGP feature contract is much simpler and only does 2 things:
1. Only accepts messages/data from the appropriate DGP framework contract (using msg.sender)
2. Stores the consensus data in a standardized way using `SSTORE` so that the blockchain can retreive and parse the data in the RLP without needing to execute the EVM

Every DGP Feature data record starts with a blockheight this consensus data should become active at. Note that old consensus data records should *NOT* be replaced or modified. This is because we may later need to reference this historical data. Although changing the historical data will not affect syncing the blockchain, but it should be kept in case it is needed.

For instance, with this Dynamic Gas Schedule data it is planned to add an additional feature so that a contract call can choose to use the gas schedule which was active when the contract was created. In order for this to be easily possible without a separate database, the historical data must remain in the RLP. This also allows the DGP to retroactively change this feature if a problem is bad enough that it is deemed necessary.

The Dynamic Gas Schedule data record will contain the following data:

1. tierStepGas0
2. tierStepGas1
3. tierStepGas2
4. tierStepGas3
5. tierStepGas4
6. tierStepGas5
7. tierStepGas6
8. tierStepGas7
9. expGas
10. expByteGas

11. sha3Gas
12. sha3WordGas
13. sloadGas
14. sstoreSetGas
15. sstoreResetGas
16. sstoreRefundGas
17. jumpdestGas
18. logGas
19. logDataGas
20. logTopicGas
21. createGas
22. callGas
23. callStipend
24. callValueTransferGas
25. callNewAccountGas
26. suicideRefundGas
27. memoryGas
28. quadCoeffDiv
29. createDataGas
30. txGas
31. txCreateGas
32. txDataZeroGas
33. txDataNonZeroGas
34. copyGas
35. extcodesizeGas
36. extcodecopyGas
37. balanceGas
38. suicideGas
39. maxCodeSize (not gas really, but related so it can stay here)

For every DGP feature, there should be an equivalent that is hardcoded into Qtum, so that the blockchain is functional without the DGP being deployed and functional. So, for Dynamic Gas Schedules there should be a default hardcoded gas schedule which is used before the DGP contract causes a new gas schedule to be used.

Activation Considerations

Although not enforced by consensus, no DGP parameter change should ever be published with less than 500 blocks before it activates. Doing so can cause unnecessary network forks and orphans. It is the responsibility of the DGP participants to ensure no proposal is approved that would be active in faster than 500 blocks.

Dynamic Gas Schedule Limits

We should determine reasonable minimums for each gas schedule item for the EVM. Some opcodes should be allowed to remain free, but also some opcodes like signature

and hashing should never be free. For right now, the minimum limits for each gas schedule item should match what is hardcoded into the default gas schedule for Qtum.

DGP Features

1. getGasSchedule
2. [reserved for getReward]
3. [reserved for getMinGasPrice]

Future Extensibility

One big need that may arise is using a significantly more complicated authorization model, such as requiring votes from the community or two separate multisig configurations or some other radically different model that what is implemented in the DGP framework contract.

It is possible to completely change the authorization model by adding a new "participant key", and then removing all other keys and reducing the maximum signatures needed to 1. This "participant key" in this case would be a smart contract which implements all of the authorization logic. Note that the authorization contract in this case would also need to handle the proposal/vote system and then simply propose and approve a matching change to the actual DGP framework contract in a single execution.

DGP Parameter Change Procedure

1. A proposal is made after discussion with the community. Unless there is an emergency issue, the activation date should be no sooner than 20,000 blocks away. If it is an emergency issue, it should be no sooner than 1000 blocks away (500 blocks to vote, 500 blocks for activation)
2. The proposal is sent to the DGP framework contract
3. The proposal is voted on
4. Assuming the vote passes, then the DGP framework sends the DGP feature contract the proposal data
5. The new DGP parameter change becomes active at the proposal activation block

What must actually be implemented

We will support multiple DGP contracts. So, we should implement a framework for supporting them.

Each DGP contract will be deployed to the blockchain and the address will be hardcoded into the blockchain.
Each DGP contract will have a specific address that contains all of the DGP data for consensus. This data should be read directly from the RLP database, and not require executing the EVM. Each DGP "record" will contain 2 items:

1. The blockheight it should become active at
2. The new parameter values to use after the specified blockheight

These records will be stored sequentially in memory as an array. There should also be a piece of data that includes the total length of the array. This data should be somewhere standardized (you may have to look up how Solidity stores arrays to figure this out).

With a DGP framework implemented that can discover the parameters stored in the RLP, it should be easily possible to use the DGP provided parameters to determine the gas schedule for the current block.

Notes

1. It should be possible to have multiple DGP parameter records
2. Caching of the parameters is fine, but the cache should be invalidated when a message is sent to the DGP contract (since the parameters could be changed)
3. If a DGP record has a retroactive blockheight (ie, record says it becomes active at block 500, and blockheight when the update is received is 1000), then it should take effect immediately and not affect previous blocks
4. Make sure to be careful of invalid data. We should gracefully ignore invalid DGP records
5. For the gas schedule DGP, if a DGP gas schedule record says to use less than default value of the template parameters (https://github.com/qtumproject/qtum-dgp/blob/master/gasSchedule-dgp-template.sol.js), then that value should be ignored and set to the value of the template parameter
6. Old DGP data should not be overwritten and should be append-only. So, when a new DGP record becomes active, no data is modified or deleted, there is only data added (except for the integer value that gives the array length)

QTUMCORE-56

Create receive_server for qtum pay

New receive_server to be created in order to support qtum ay from external applications.

Server to create addresses for all transactions up on request from the external app. Also to transfer all money to the registered address.

QTUMCORE-57

1. # Create mobile sdk for qtum pay

   Create mobile sdk (for Android and iOS) to support in-app purchases being paid by qtum.

Mobile sdk to do next actions:

1. send request to the receive_server with parameters to generate new address
2. send request to the receive_server to check status for required transaction
3. restore in-app purchases in case of re-installing the app or switching devices for user

In addition implement pay by qtum that will open details where to transfer money and option to open qtum wallet by mistake added here. moved to products board

QTUMCORE-58

Add additional AAL and Condensing TX validation rules

We have just implemented the AAL. However, it's validation rules are not completely secure yet. We should do the following when receiving a block with condensing executions:

Implementing this will allow us to remove previous OP_TXHASH one by one checks (no need for double execution), and will allow the validation of all VM future tx types (including the recent validation condensing transaction ITD).

0. First make sure stateRoot and utxoRoot hashes are part of the block hash computation.

1. Process block using checkblock, and all other checks needed before validating transaction scripts and contract execution
2. Create a completely new block, copying all header info from the original block
3. Add coinbase and stake transactions from original block
4. Add 1 transaction at a time from original block, executing contracts as needed
5. Any expected OP_TXHASH transactions should be added to the block in-order of creation. If there is an unexpected OP_TXHASH tx in the original block, it should be rejected (this might be changed later for soft-fork compatibility)
6. Process every transaction until none remain
7. Create new merklehash and place in blockheader
8. Compute state root hashes etc EVM state data
9. Compare the new block's hash to the old block's hash
10. If the new block's hash matches, then accept it. Otherwise, reject the block

Pseudo-code:

```
block = receiveBlock();
CheckBlock(block)
testBlock = new Block();
testBlock.header=block.header
testBlock.MerkelRoot=0; //make 0 because no transactions in block yet
foreach(tx in block){
    if(tx.hasExec()){
        testblock.Add(tx) //add contract tx to block
        result = tx.exec();
        if(result.hasOpHashTx()){
            foreach(opHashTx in result){
                testblock.Add(opHashTx); //add resulting condensing tx from
contract execution (I think there can be more than 1 if a single tx has
multiple EVM execs?)
            }
        }
    }else if(tx.isContractSpend()){
        //don't add spends
    }else{
        testblock.Add(tx) //add any other tx type, non-standard, pubkeyhash,
etc
    }
}
testBlock.calculateMerkelRoot();
testBlock.calculateStateRoots();
assert(testBlock.hash() == block.hash())
```

QTUMCORE-59

Validation of correctness of condensing transaction

After adding to the block a condensing transaction by miner there is a risk of vouts replacement.

As a result:
Users' balances can be lost.
Data in dbUTXO and dbState will be corrupted.

Environment
Modifying miner the way that all balances from the condensing transaction are sent to miner.

Test Steps
Create contract:
contract Temp {
function () payable {}
}
Mining created block.
Adding balances to created contract.
Mining created block.

Expected Results
Block not accepted.

QTUMCORE-60

Gas Schedule DGP implementation details

Below work needs to be done in the submodule, please try to keep it concise and clean, with as little as possible changes to the main code, for example by using functions/classes inside the evm code and put the definitions in a new file. This way we can keep the EVM code clean for future updates.

1- create a new qtum genesis file: libethashseal/genesis/qtumMainNetwork.cpp

the file should be based on: libethashseal/genesis/mainNetwork.cpp

with below changes:

First we change constants name:

static dev::h256 const c_genesisStateRootQtumMainNetwork(""); // stateRootHash should be placed here after its computation
static std::string const c_genesisInfoQtumMainNetwork = std::string() +

then params section:

```
"homsteadForkBlock": "0x0",
                "daoHardforkBlock": "0xffffffffffffff",
                "EIP150ForkBlock": "0x0",
                "EIP158ForkBlock": "0x0",
                "registrar": ""
```

also in genesis section:

```
"extraData":
"0x5174756d4d61696e4e6574c9987fd35877cdbbbb84ffeb5315ab1f86c21398c0",
```

and in accounts section:

keep the top 4 accounts (precompiled contracts) **and remove all the rest.**

then we add 5 accounts: all instances of dgp-template.sol.js (https://github.com/qtumproject/qtum-dgp)


```
"0000000000000000000000000000000000000080": {
"code": "0x6060604052361561011057600357c010000000....",
"storage":
"{"c2575a0e9e593c00f959f8c92f12db2869c3395a3b0502d05e2516446f71f85b":
{"0000000000000000000000000000000000000000000000000000000000000003":
"000000000000000000000000000000000000000000000000000000000000001e"}}"
},
"0000000000000000000000000000000000000081": {
"code": "0x6060604052361561011057600357c010000000....",
"storage":
"{"c2575a0e9e593c00f959f8c92f12db2869c3395a3b0502d05e2516446f71f85b":
{"0000000000000000000000000000000000000000000000000000000000000003":
"000000000000000000000000000000000000000000000000000000000000001e"}}"
},
"0000000000000000000000000000000000000082": {
"code": "0x6060604052361561011057600357c010000000....",
"storage":
"{"c2575a0e9e593c00f959f8c92f12db2869c3395a3b0502d05e2516446f71f85b":
{"0000000000000000000000000000000000000000000000000000000000000003":
"000000000000000000000000000000000000000000000000000000000000001e"}}"
},
"0000000000000000000000000000000000000083": {
"code": "0x6060604052361561011057600357c010000000....",
"storage":
"{"c2575a0e9e593c00f959f8c92f12db2869c3395a3b0502d05e2516446f71f85b":
{"0000000000000000000000000000000000000000000000000000000000000003":
"000000000000000000000000000000000000000000000000000000000000001e"}}"
},
"0000000000000000000000000000000000000084": {
"code": "0x6060604052361561011057600357c010000000....",
"storage":
"{"c2575a0e9e593c00f959f8c92f12db2869c3395a3b0502d05e2516446f71f85b":
{"0000000000000000000000000000000000000000000000000000000000000003":
"000000000000000000000000000000000000000000000000000000000000001e"}}"
}
```


those will be used for gas schedule, minimum gas price, and block size ... this story only focuses on the first one.

the code should contain the deployed code of the DGP contract (can be obtained using getacocuntinfo, remember to add 0x)

the format of storage and possibly code might need to be slightly different, refer to libethereum/Account.cpp lines 75 to 106 for details about the format

the objective here is to have both contracts as part of the genesis state and can be called from the rpc ...

also don't forget to update c_genesisStateRootQtumMainNetwork with the new stateRootHash, and also the core genesis stateRootHash as well as tests hash

2. related changes (indications)

in file: /libethashseal/GenesisInfo.h

add qtumMainNetwork = 9, ///QTUM Homestead + EIP150 + EIP158 Rules active from block 0 to enum class Network

in file: /libethashseal/GenesisInfo.cpp

add

#include "genesis/qtumMainNetwork.cpp"

add

```
case Network::qtumMainNetwork: return c_genesisInfoQtumMainNetwork;
```

to std::string const& dev::eth::genesisInfo(Network _n) cases

and

```
case Network::qtumMainNetwork: return
c_genesisStateRootQtumMainNetwork;
```

to h256 const& dev::eth::genesisStateRoot(Network _n) cases

3. replace dev::eth::Network::HomesteadTest with dev::eth::Network::qtumMainNetwork in validation.cpp so we use the new genesis for Qtum, as well as anywhere else needed.

4. Before moving forward, make sure the submodule compiles, and the core compiles and passes tests. And that both contracts are callabe with their address

5. Now for the fun part, we need to implement the core functionality of the DGP, which is read Params from the contracts:

the way to get params is to read storage from address: 0000000000000000000000000000000000000080 with globalState->storage(addrAccount) or a better solution

first we get storage from 0000000000000000000000000000000000000080 and reconstruct paramsHistory array (see DGP contract, reconstruction can be done using existing functions to parse storage or create one)

then we create a function that will have a blockNumber as parameter (the function wil do the following):

1. the function will parse the array and return the current address of gas Schedule contract (similar to getParamsForBlock in the solidity contract)

a good place to implement this function call is before line 63 of the file: libethcore/SealEngine.cpp in function: EVMSchedule const& SealEngineBase::evmSchedule(EnvInfo const& _envInfo) const

let's call this function qtumDGP->getGasSchedule(u256 blockHeight) (let's create a separate class in new files .h, .cpp to manage all DGP future functions) and call it qtumDGP.h qtumDGP.cpp ...

if the address returned is not null (!=0000000000000000000000000000000000000000)

- the function will then get the storage at the returned address using globalState->storage(scheduleAddress) or a better solution, then parse it to get the gas values for each op, then return the parsed schedule

  if the address returned is null (=0000000000000000000000000000000000000000)

- Use the default hardcoded schedule in the core code, the function will return false and execution will continue as default

  the first test will return a null address, we then need to deploy dgp-template.sol.js contract, add an admin to 0000000000000000000000000000000000000080 and add a new schedule address (deployed contract address) to 0000000000000000000000000000000000000080
  which shall be returned if the function is called again.

  please check both DGP contracts by deploying them and adding different schedules at different blocks, and analysing storage changes to get an idea how the storage should be parsed.

  sample main DGP storage:

```
"storage": {

"290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e563": {

"0000000000000000000000000000000000000000000000000000000000000000":
"0000000000000000000000000000000000000000000000000000000000000005"
   },

"45991fce5fc3033a0031207ac65e4c413069a0cd6e6bc7664254f8c4341cfe66": {

"290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e56a":
"000000000000000000000009c63b222d9db1de070e45802d7fd636cb5da922a"
   },
```

```
"510e4e770828ddbf7f7b00ab00a9f6adaf81c0dc9cc85f1f8249c256942d61d9": {

"290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e563":
"00000000000000000000000000000000000000000000000000000000000000ef"
   },

"51bdce570797d7347ee2c632abdbe21de6c1cddf1026b9348df268225cf35eed": {

"290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e56b":
"00000000000000000000000000000000000000000000000000000000000000f3"
   },

"5306a7ea1091503364459f70885dc372117f70834621ea9300aa244571124d0a": {

"290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e568":
"000000000000000000000008c63b222d9db1de070e45802d7fd636cb5da922a"
   },

"63d75db57ae45c3799740c3cd8dcee96a498324843d79ae390adc81d74b52f13": {

"290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e565":
"00000000000000000000000000000000000000000000000000000000000000f0"
   },

"68ebfc8da80bd809b12832608f406ef96007b3a567d97edcfc62f0f6f6a6d8fa": {

"290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e566":
"000000000000000000000007c63b222d9db1de070e45802d7fd636cb5da922a"
   },

"6c13d8c1c5df666ea9ca2a428504a3776c8ca01021c3a1524ca7d765f600979a": {

"290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e564":
"000000000000000000000006c63b222d9db1de070e45802d7fd636cb5da922a"
   },

"7b5fcc8f73196524ea5f04c38888c2f09c6cbef411cb31e259d35b56e3d0047b": {

"290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e56c":
"000000000000000000000001063b222d9db1de070e45802d7fd636cb5da922a"
   },

"8a35acfbc15ff81a39ae7d344fd709f28e8600b4aa8c65c6b64bfe7fe36bd19b": {
```

```
"0000000000000000000000000000000000000000000000000000000000000004":
"0000000000000000000000000000000000000000000000000000000000000001"
   },

"9c418048a637d1641c6d732dd38174732bbf7b47a1cf6d5f65895384518b07d9": {

"290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e567":
"00000000000000000000000000000000000000000000000000000000000000f1"
   },

"aaab8540682e3a537d17674663ea013e92c83fdd69958f314b4521edb3b76f1a": {

"290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e569":
"00000000000000000000000000000000000000000000000000000000000000f2"
   },

"b10e2d527612073b26eecdfd717e6a320cf44b4afac2b0732d9fcbe2b7fa0cf6": {

"0000000000000000000000000000000000000000000000000000000000000001":
"0000000000000000000000000000000000000000000000000000000000000001"
   },

"b5d9d894133a730aa651ef62d26b0ffa846233c74177a591a4a896adfda97d22": {

"b10e2d527612073b26eecdfd717e6a320cf44b4afac2b0732d9fcbe2b7fa0cf6":
"000000000000000000000008c53de5f85032a9cba46ccbdc2b7c2598ec58f42"
   },

"c2575a0e9e593c00f959f8c92f12db2869c3395a3b0502d05e2516446f71f85b": {

"0000000000000000000000000000000000000000000000000000000000000003":
"000000000000000000000000000000000000000000000000000000000000001e"
   }
 },
```

you can see that the paramsHistory array elements are detectable:

```
  "63d75db57ae45c3799740c3cd8dcee96a498324843d79ae390adc81d74b52f13": {
    "290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e565":
"00000000000000000000000000000000000000000000000000000000000000f0"
   },
   "68ebfc8da80bd809b12832608f406ef96007b3a567d97edcfc62f0f6f6a6d8fa": {
    "290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e566":
"000000000000000000000007c63b222d9db1de070e45802d7fd636cb5da922a"
```

those shows a paramsHistory array element which is a struct:

```
        struct paramsInstance{
                uint blockHeight;
                address paramsAddress;
        }
```

we can see all elements of the array share the same key number, which is incremented by 1

sample schedule state is simpler:

```
  "storage": {
    "290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e563": {
      "0000000000000000000000000000000000000000000000000000000000000000":
"0000000a0000000a0000000a0000000a0000000a0000000a0000000a0000000a"
    },
    "405787fa12a823e0f2b7631cc41b3ba8828b3321ca811111fa75cd3aa3bb5ace": {
      "0000000000000000000000000000000000000000000000000000000000000002":
"00002328000008fc0000002800007d000000017700000008000017700000001"
    },
    "8a35acfbc15ff81a39ae7d344fd709f28e8600b4aa8c65c6b64bfe7fe36bd19b": {
      "0000000000000000000000000000000000000000000000000000000000000004":
"00000000ffffffff000000000000014000000140000001400000003000000044"
    },
    "b10e2d527612073b26eecdfd717e6a320cf44b4afac2b0732d9fcbe2b7fa0cf6": {
      "0000000000000000000000000000000000000000000000000000000000000001":
"00003a980000138800004e2000000320000000600000001e0000000a0000000a"
    },
    "c2575a0e9e593c00f959f8c92f12db2869c3395a3b0502d05e2516446f71f85b": {
      "0000000000000000000000000000000000000000000000000000000000000003":
"000000040000cf08000052080000000c8000002000000000300005dc0000061a8"
    }
  },
```

what we need here is to sort the key, 0 to 4, then parse the gas values which are represented as 4 bytes starting from the right, so above the first gas value is 0000000a, and the 9th is also 0000000a

fell free to discuss ideas you might have to implement this better.

QTUMCORE-61

Implement Mutualized Proof Of Stake Block Reward System

Mutualized Proof Of Stake (MPoS) is a new reward system designed to make DoS attacks on Qtum more expensive for the attacker.

Goals

1. Prevent malicious miners from attacking the network for free by constructing expensive to validate blocks, and then receiving all of the gas fees back to themselves through the mining process
2. Help to make it more difficult and expensive for an attacker to DoS the network

Procedure

1. When a staker mines a block, he receives only a small portion of the PoS reward and gas fees. The rest of the reward and fees are shared with 9 other people.
2. When a staker mines a block, his stake script (staketx.vout[0]) is registered to receive a share of the reward, lasting 10 blocks, 500 blocks from when the block was mined
3. Thus, every block there will be 10 reward recipients. The creator of the block, and 9 "mutual stakers".
4. After 9 blocks of shared rewards, the staker's script will be removed, and another will be added to replace it
5. If a stake script has mined more than 1 block in a 10 block period, then there can be a case where he receives 2x the share. However, once the earliest stake script instance exceeds 510 blocks from it's mined block, it is dropped and the reward drops to normal. Identical stake scripts should not be combined into a single UTXO, the rewards should be duplicated
6. In order to prevent a malicious or careless staker from mining contracts that have either a free, or very cheap gas price, a "Minimum Gas Price DGP" feature will be created in a later story, so that a minimum gas price can be made. This would be increased in times of exploits and crisis, and decreased in times where the gas price becomes repressive and makes using contracts too expensive.

Example

1. There is a staker, X, who mines blocks 1000
2. X receives 1/10th of the block reward and gas fees for the block. The staking transaction includes at least 9 other outputs from the other mutual stakers
3. X mines another block at height 1004
4. X receives 1/10th of the block reward and gas fees for the block. The staking transaction includes at least 9 other outputs from the other mutual stakers
5. X then doesn't mine anymore blocks for the duration of the example
6. At blocks 1500, 1501, 1502, and 14503, X receives a UTXO for each block for 1/10th of the block reward and gas fees for the created blocks, along with the block creator and

8 other stakers
7. At blocks 1504-1509, X receives 2 UTXOs for each block for 1/10th of the block reward and gas fees for the created blocks, along with the block creator and 7 other stakers (even though X is a single staker, he receives 2 UTXOs for mining 2 blocks in a small span of time)
8. At blocks 1510-1513, X receives 1 UTXO for each block for 1/10th of the block reward and gas fees for the created blocks, along with the block creator and 8 other stakers.
9. X's MPoS period is complete, he will not receive anymore MPoS rewards until he mines another block

Consensus Rules

- The block reward sent to the MPoS stakers must come from the staking block
- Thus, the stake transaction must contain at least 10 outputs, 1 for the creator, and 9 for the MPoS stakers
- The first 10 outputs of the stake transaction are consensus critical and must be in exactly the correct order based on the MPoS staker blockheight and with the creator being the first output.
- The stake transaction can contain additional outputs, such as for splitting a large staking UTXO into multiple UTXOs. After the 10th output, there should not be any special consensus rules for the outputs so long as the output value does not cause the transaction to exceed the input value+block reward+fees
- The first 500 blocks of PoS does not use MPoS, and so the block creator gets the full reward without using this logic

Note that the 0th output of the staking transaction should be empty (so that it is detected as a staking transaction). This story breaks some of the "mutlisig staking" (misleading name) functionality from Blackcoin. We can fix this in a later story

Options

QTUMCORE-62

Remove timestamp field from transactions

Because of our implementation of PoSv3, there is no strong purpose or usecase for timestamps in transactions, and in fact it can make some things more complicated like micropayment channels.

The field should be removed, all rules around it's enforcement should be removed, and in the PoS kernel hash instead of using the transaction timestamp, the block timestamp should be used (there was a consensus rule enforcing these to be equal, so no logic is

actually changed). If there are any security concerns beyond the coinstake transaction being used in the kernel hash, they should be brought up and discussed. (as far as I know, there are none)

QTUMCORE-63

Properly describe contract transactions in Qt transaction list

Right now if you deploy or send to contracts there is no description or indication what that transaction was. Additionally, there should be an indication that "mined transactions" from AAL refunds are a refund and not the wallet automatically mining by itself.

QTUMCORE-71

Create tests for QtumDGP

Qtum 测试网络预计 6 月份发布，感谢您的支持！

如果有兴趣加入 Qtum，请发简历到 job@qtum.org