Concrete Architecture of Bitcoin Core

Group 20 - GoDaddy

Junhui Zong
Brian Cho
Julian Wei
Yuchen Li
Ye Shen
Arsh Kochhar

Abstract

This report discusses the concrete architecture of Bitcoin Core and compares it to our original conceptual architecture to propose a modified one. The concrete architecture was derived using Understand. The modified conceptual architecture includes two new modules: *API* and *Utilities*, which are essential for handling and processing blocks and transactions within the blockchain network. Unexpected dependencies between subsystems are highlighted, such as *Peer Discovery* ↔ *Storage Engine*, *Validation Engine* → *Mempool*, *Wallet* → *Validation Engine*, *Wallet* → *Peer Discovery*, *Storage Engine* → *Validation Engine*, and *Peer Discovery* ↔*Validation Engine*.

Additionally, the report explores the *Miner* subsystem. It was found that it contains only a single subcomponent called BlockAssembler. After performing reflexion analysis, it was found that Miner lacks mining functionalities, which is in contrast to our conceptual knowledge of the subsystem. It was found that the actual mining occurs in the RPC subsystem's mining.cpp file.

Introduction and Overview

Blockchain technology, exemplified by Bitcoin, has risen in popularity in recent years. Among various software implementations on the Bitcoin network, Bitcoin Core stands out, and this report investigates its concrete architecture and subsystems.

The report starts off by utilizing the top-level dependency diagram to help generate a modified conceptual architecture that highlights unexpected dependencies and absent components that the team was not able to identify prior. After analyzing the concrete architecture further, the report identifies the concrete subsystems, and dives deeper into their interactions with each other. This leads the report into the reflection analysis which clarifies the new modules seen in the concrete subsystems as well as the unexpected dependencies that were not previously identified during the conceptual system analysis.

The report then delves into the *Miner* subsystem in great detail. Its subsystems, functionalities, interactions, and divergences are discussed.

A couple use cases are presented which showcase interactions between subsystems to illustrate how they bring value to the concrete architecture of the system as a whole.

Derivation Process

For the top-level architecture, we derived our concrete architecture graph by using the Understand graphical tool. In Understand, all the available dependencies are demonstrated for the architecture. In this way, compared to our original conceptual architecture, we found unexpected dependencies. Furthermore, by using the

dependency browser in Understand, we were available to analyze the functions/constants in dependencies. We eventually complete our concrete architecture diagram by those steps.

We derived our *Miner* dependency graph from two main sources of information, Understand's graphical tools and the source code in miner.cpp. The graphical tools allowed us to see miner.cpp's dependencies and where these interactions occurred in the source code. From the tool, we looked into the source code and determined what each relation did. This was possible due to the small size of the miner subsystem. From this, we found that miner.cpp only had one major component, the BlockAssembler. These findings then came together to influence our diagram in the end.

Top-Level Architecture

Dependency Diagram

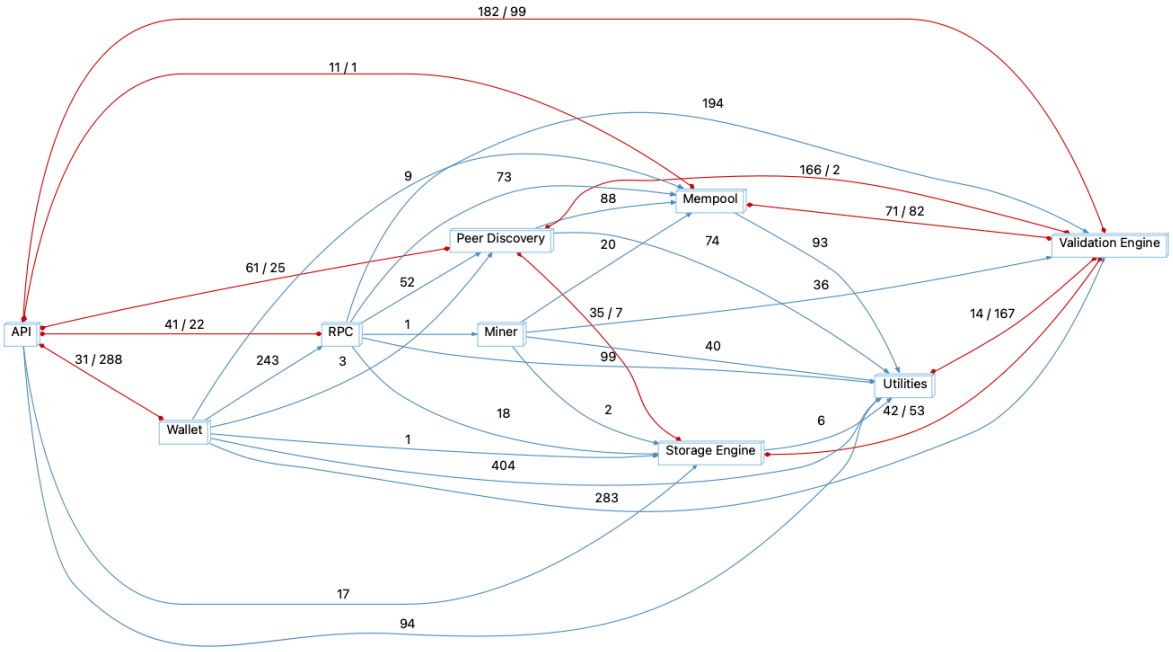

Figure 1: Dependency diagram based generated from Understand

We used Understand to generate this dependency diagram, which indicates all the dependencies (both known and unexpected) in this architecture.

Concrete Architecture

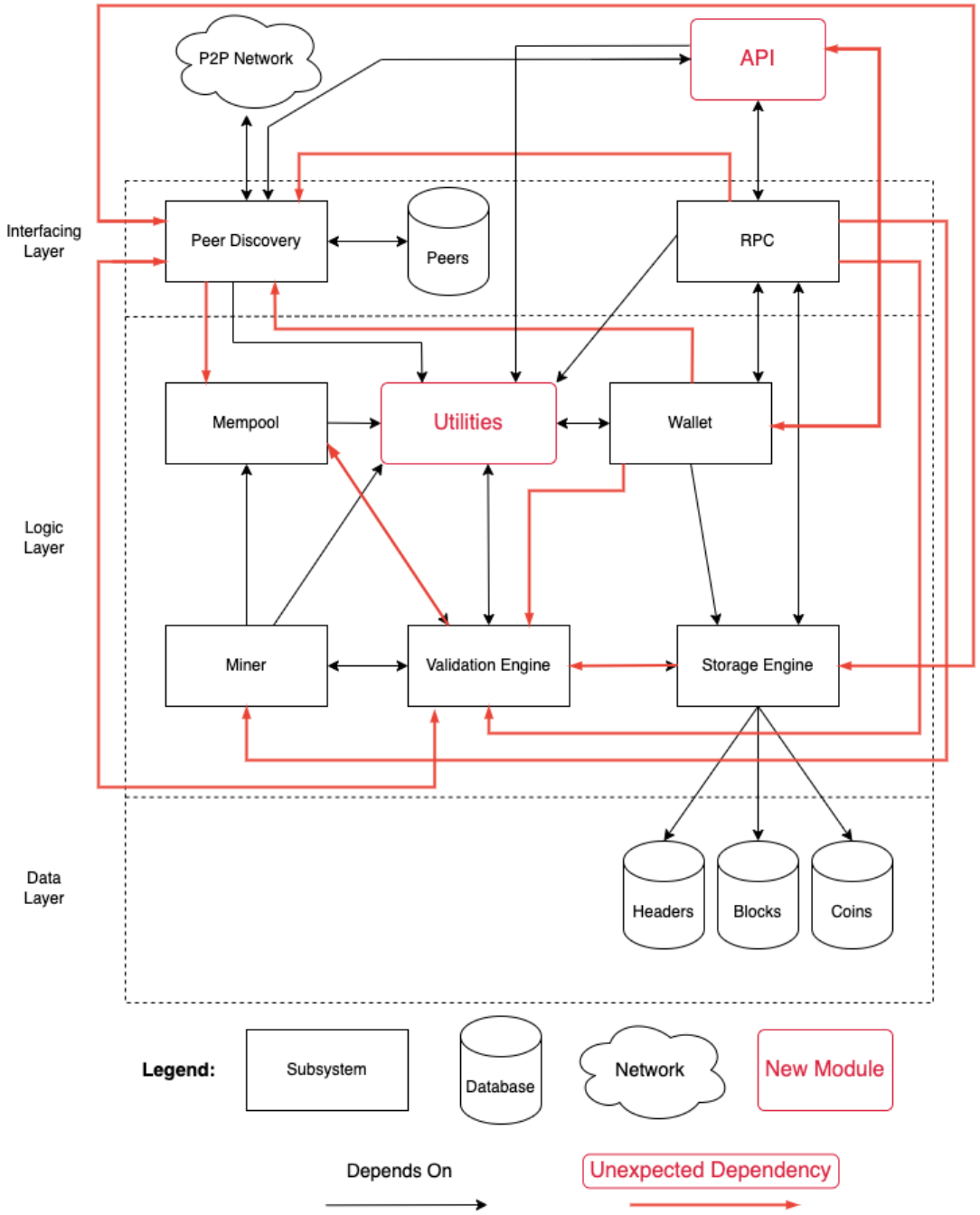

Figure 2: Concrete architecture built based on Understand

The conceptual architecture has been modified to be our concrete architecture. It has two new modules: *API* and *Utilities*. And it is noticeable that *Connection Manager* is no longer counted in the concrete architecture. The concrete architecture still uses the P2P, Client-Server, and Layered style. As mentioned later on, some of the subsystems are found to use the pub-and-sub as well as the interpreter style.

Concrete Subsystems

Mempool

The *Mempool* stores unconfirmed transactions in “txmempool.cpp” waiting to be validated by a miner and added to the blockchain. Transactions are verified using “validation.cpp” in *Validation Engine* before being added to the queue.

Wallet

The Bitcoin Core *Wallet* is a software component that stores, manages, and tracks bitcoin ownership. It generates public keys - CPubKey in src/pubkey.h and private keys - CKey at src/key.h included in bitcoin-wallet.cpp, where the public key identifies the wallet, and the private key is used to sign transactions using Sign() and control access to the wallet's bitcoins. The wallet communicates with a Bitcoin Core node using an *RPC* interface - src/rpc/mempool.cpp to validate transactions using “validation.cpp” in *Validation Engine* and broadcast them to the network.

Peer Discovery

The *Peer Discovery* subsystem as discussed in the conceptual architecture has two main responsibilities, the acquisition and storage of new ip addresses and the processing of data inflow and outflow. For the first responsibility, file addrman.cpp serves as the central controller, offering address storage manipulation through functions Find(), Create(), Delete(), AddSingle(). As for data processing, file net_processing plays an important role in both inflow and outflow. Inflow involves converting network messages into workable data and can be seen in the function ProcessMessage(). Outflow is the broadcasting of data from the node to the network and can be seen in functions like SendMessage().

RPC

The *RPC* subsystem provides a set of commands that can be used by external applications to interact with the Bitcoin Core program. rpc/server.cpp handles the bootup and shutdown of *RPC* as well as provides an interface to pass in commands for execution.

Miner

The *Miner* is responsible for dealing with the creation of new blocks. It accesses txmempool.cpp of *Mempool* to gain access to new transactions and uses validation.cpp of *Validation Engine* to verify the newly created block.

Validation Engine

The *Validation Engine* verifies the validity of blocks and transactions on the Bitcoin Work. It first checks the new blocks received by the Bitcoin Core, and determines

whether they are valid based on the network rules. It interacts with blockstorage.h of *Storage Engine* to read from disk as well as verify the database.

Storage Engine

The *Storage Engine* serves as a database management subsystem which deals with the storage of transaction data in src/node/blockstorage.cpp. It features transaction indexing that speeds up the process of verifying new blocks in the blockchain using “validation.cpp” in *Validation Engine*.

Utilities

Utilities is a subsystem that aids other subsystems in their implementation. This explains the many dependencies on it. *Utilities* is mostly made up of the util folder and the primitives folder which contains block.h and transaction.h to represent blocks and transactions. It also contains the scripts folder which is used for representing transaction data. Script uses an interpreter style using script/interpreter.cpp to read Bitcoin Scripts.

API

API provides an interface for other programs to interact with Bitcoin Core. It is made up of the interfaces folder. Most of the interfaces provided for the node is in interfaces/node.h.

Qt is used for the GUI component. Notably, it uses pub-and-sub style to receive notifications from interfaces/node.h to qt/bitcoingui.h.

Reflection Analysis on the Top-Level Architecture

New Modules

Two new components that were not present in our conceptual architecture were identified. These are *API* and *Utilities*. As stated earlier, *API* provides an interface for the user or external software applications to interact with Bitcoin Core, and *Utilities* provides utilities for other subcomponents to use to process blocks, transactions, and perform other actions.

Unexpected Dependencies

Peer Discovery ↔ Storage Engine

There is a bidirectional dependency between “blockstorage.cpp” and “protocol.h”. This is because “protocol.h” requires functions from “blockstorage.cpp”. There is a potential situation that “protocol.h” defines some protocols for interacting with a storage system, and therefore “blockstorage.cpp” provides the implementation of that protocol.

Peer Discovery → Mempool

There is a dependency from “net_processing.cpp” to both “mempool_entry.h” and “txmempool.h”. This allows Peer Discovery to complete tasks related to verifying transactions and determining whether to accept incoming transactions. In addition, “m_time” constant and functions including “exists”, “DynamicMemoryUsage”, “GetMinFee” are specified in “mempool_entry.h” and “txmempool.h” and are required by “net_processing.cpp” for their features.

Validation Engine → Mempool

There is a dependency from “validation.cpp” to “mempool_entry.h” so that *Validation Engine* can check *Mempool*’s sequence locks, perform pre-checks, and potentially other operations that involve handling *Mempool* transactions.

Wallet → Validation Engine

There is a dependency from “wallet_create_tx.cpp” to “validation.h”. That is because "wallet_create_tx.cpp" needs "validation.h" for some common utility functions related to transactions and blocks. Additionally, "generateFakeBlock" and "getTip" functions in "wallet_create_tx.cpp" use the "cs_main" variable, which is declared in "validation.h".

Wallet → Peer Discovery

There is a dependency from “init.cpp” to “net.h” so it can initialize *Wallet* to connect to other nodes in the network in order to function properly. Furthermore, “DEFAULT_BLOCKSONLY” constant is defined in “net.h” and required by “init.cpp” for its features.

Storage Engine → Validation Engine

There is a dependency from “blockstorage.cpp” to “validation.h” because it uses "MIN_BLOCKS_TO_KEEP" constant, "AbortNode" function, and various utility functions related to block validation and management, including "LoadGenesisBlock", "ActiveChainstate", and "LoadMempool" which are all defined in "validation.h" and needed by "blockstorage.cpp" for its functionality.

Peer Discovery ↔Validation Engine

There is a bidirectional dependency from “validation.h” to both “net_processing.cpp” and “transaction.cpp”. That is probably because “net_processing.cpp”and “transaction.cpp” use functions from “validation.h” to get the validation result and state of a block. The “validation.h” header file contains the definition of functions that related to peer discovery. Therefore, “validation.h” is included in “net_processing.cpp” and “transaction.cpp” to provide the necessary functions for the validation information in the peer discovery process.

RPC → Peer Discovery

There is a dependency from “src/rpc/net.cpp” to “src/net.cpp” so that the RPC command of establishing a connection with the peer can be made using *Peer Discovery*.

API ↔ Wallet

There is a bidirectional dependency from “validationinterface.h” to both “wallet_balance.cpp” and “wallet_loading.cpp”, from “interface_ui.h” and “walletinitinterface.h” to “init.cpp”. That is probably because “wallet_balance.cpp”, “wallet_loading.cpp” and “init.cpp” call or override functions related to Wallet management that’s defined in “interface_ui.h” and “walletinitinterface.h”. Therefore, “wallet_balance.cpp” and “wallet_loading.cpp” include “validationinterface.h” and either call or override function from it for their functionality.

RPC → Miner

There is a dependency from “rpc/mining.cpp” to “node/miner.h” so that the RPC command of mining new blocks can use *Miner* to initialize a new block to start mining on.

Missing Module

The *Connection Manager* module was not taken into account in the concrete architecture because it was thought to manage connections among components, which isn’t something that happens in the concrete architecture.

Inner Architecture: Miner

In this section we will be discussing the *Miner* subsystem and its interactions. The *Miner* subsystem has only two files, node/miner.cpp, and its header file node/miner.h.

File node/miner.cpp and node/miner.h facilitates the assembly of blocks and unconfirmed transactions into a block template which could then be put through the validation process. This assembly of blocks is done through the public function CreateNewBlock(), defined in the class *Block Assembler*, which is the *Miner*’s only major component. To create a new block, CreateNewBlock() accesses the *Mempool* and starts by checking which transactions are viable for block inclusion. Checks include the onlyUnconfirmed() and TestPackageTransaction() methods to test whether the transaction packages are unconfirmed and final, respectively. After passing the checks, transactions are added to the new block through the function *AddToBlock()*. This new block (template), now with all the unconfirmed transactions can now be validated and potentially submitted by the *RPC* and broadcasted out into the bitcoin network.

In the assembly of the template, *BlockAssembler* interacts with four other subsystems: *Mempool, RPC*, *Validation Engine,* and *Utilities.*

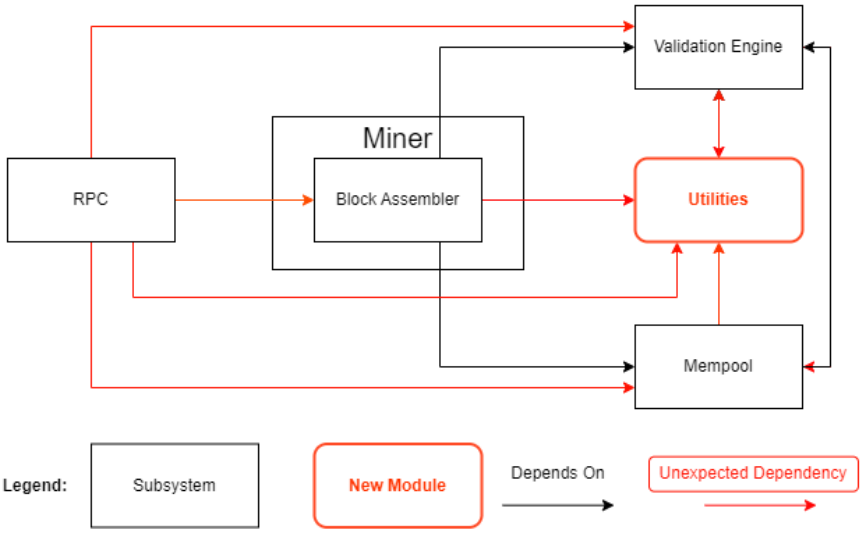

Figure 3: Concrete architecture of *Miner*

BlockAssembler depends on *Mempool* to supply unconfirmed transactions for inclusion in the template. *BlockAssembler* has several functions (AddToBlock(), SortForBlock()) which takes the *Mempool* object (CTXMempool) as an argument to perform checks and searches.

For *RPC*, rpc/mining.cpp depends on miner.cpp as it calls *Block Assembler* to initialize the assembly procedure. This can be seen in the RPC command generateblock() which attempts to call miner.cpp to generate a new block.

For the *Validation Engine*, miner.cpp depends on validation.cpp (component of *Validation Engine*) through a function call to GetBlockSubsidy() for calculations involving the amount of subsidies the user gets for mining a new block. There is also a dependency from miner.cpp to pow.cpp (another component of *Validation Engine*) through GetNextWorkRequired(), which attempts to estimate the amount of work required so that miner.cpp can correctly update the block creation time.

Block Assembler interacts with *Utilities* to access commonly used tools and data types.

Reflexion Analysis of Miner

Oddly, in contrast to our conceptual knowledge of the subsystem, *Miner* does not have any functionalities related to mining the blocks (finding a valid proof of work). The component that is responsible for this is *RPC* where the mining implementation is placed in rpc/mining.cpp.

The reason for this is not all too clear. Part of it is due to the miner file being renamed and moved frequently. However, we do know that there did exist a miner.cpp file that had mining functionalities (commit d247a5d1). Later on, commit 6b04508e “[Introduced a] separate ‘generate’ RPC call” with no additional commit message. Then, commit 8d1de43f removed the BitcoinMiner from miner.cpp as it is only useful for testing.

Somewhere along the way, BitcoinMiner transitioned from being the core miner to a testing miner. Exactly why the entire miner functionality was put inside the ‘generate’ RPC is unknown since the developer did not provide further comments. A guess would be that since *Miner* would only be called by an RPC command, the rationale would have been to put the whole mining functionality inside the RPC command directly.

Another absence we were able to identify was *Miner* checking the proof of work using *Validation Engine*. This does not happen since the actual mining happens in RPC.

Unexpectedly, Miner uses *Validation Engine* to get the block subsidy using GetBlockSubsidy() in validation.cpp. This method is in the Validation *Engine* as the output differs depending on the consensus parameters.

In terms of dependency divergences, there is just one which is the dependency from RPC to Miner. This dependency is due to having to create a new block using CreateNewBlock() using the *Block Assembler*. The reason for this divergence is actually not unexpected; we simply forgot to state this as a dependency for our conceptual architecture. That being said, the dependency is caused only for the aspect of assembling the blocks and not mining the blocks.

Use Cases

Use Case #1: Executing an RPC Command

When Bitcoin Core is booted, an initialization process occurs in init.cpp of the *Initializer* subcomponent of *Utilities* to set everything up. AppInitMain() is responsible for this. *Initializer* calls RegisterAllCoreRPCCommands() in register.h to store all the RPC commands inside the *CRPCTable*, which is a look-up table for an RPC command. *CRPCTable* and *CRPCCommand* are implemented inside rpc/server.cpp that is in *RPC*.

While the program is running, the user can execute a command by passing in the command name to execute as well as some parameters through the console (qt/rpcconsole.cpp). This data is parsed in the *RPCParseCommandLine()* method and then the console invokes *executeRPC()* from *API* (node/interfaces.cpp).

The node wraps the arguments into a *JSONRPCRequest* object and invokes the *execute()* method on the *CRPCTable* to execute the command. The *execute()* method looks inside a hashmap using the command name as the key. The hashmap returns a list of commands to pass into the *ExecuteCommands()* method. The *ExecuteCommands()* method is odd in that it has a loop for all the commands in the list, but it terminates after executing the first one. Nevertheless, *ExecuteCommands()* invokes *ExecuteCommand()* to execute the commands in the list.

ExecuteCommand() takes the *CRPCCommand* that was passed in and invokes the *actor()* method. The *actor()* method is a lambda function which invokes the method that was stored inside *CRPCCommand* (as a function pointer). For example, it might have stored the *generateblock()* method implemented inside rpc/mining.cpp. These methods return an *RPCHelpMan* object in which *actor()* invokes *HandleRequest()* on it (rpc/server.h). The *HandleRequest*() method takes the *JSONRPCRequest* and executes a function that was stored inside *RPCHelpMan*. This function is what executes tasks and is what we would view as the actual command. The name of this function (pointer) is *m_fun*.

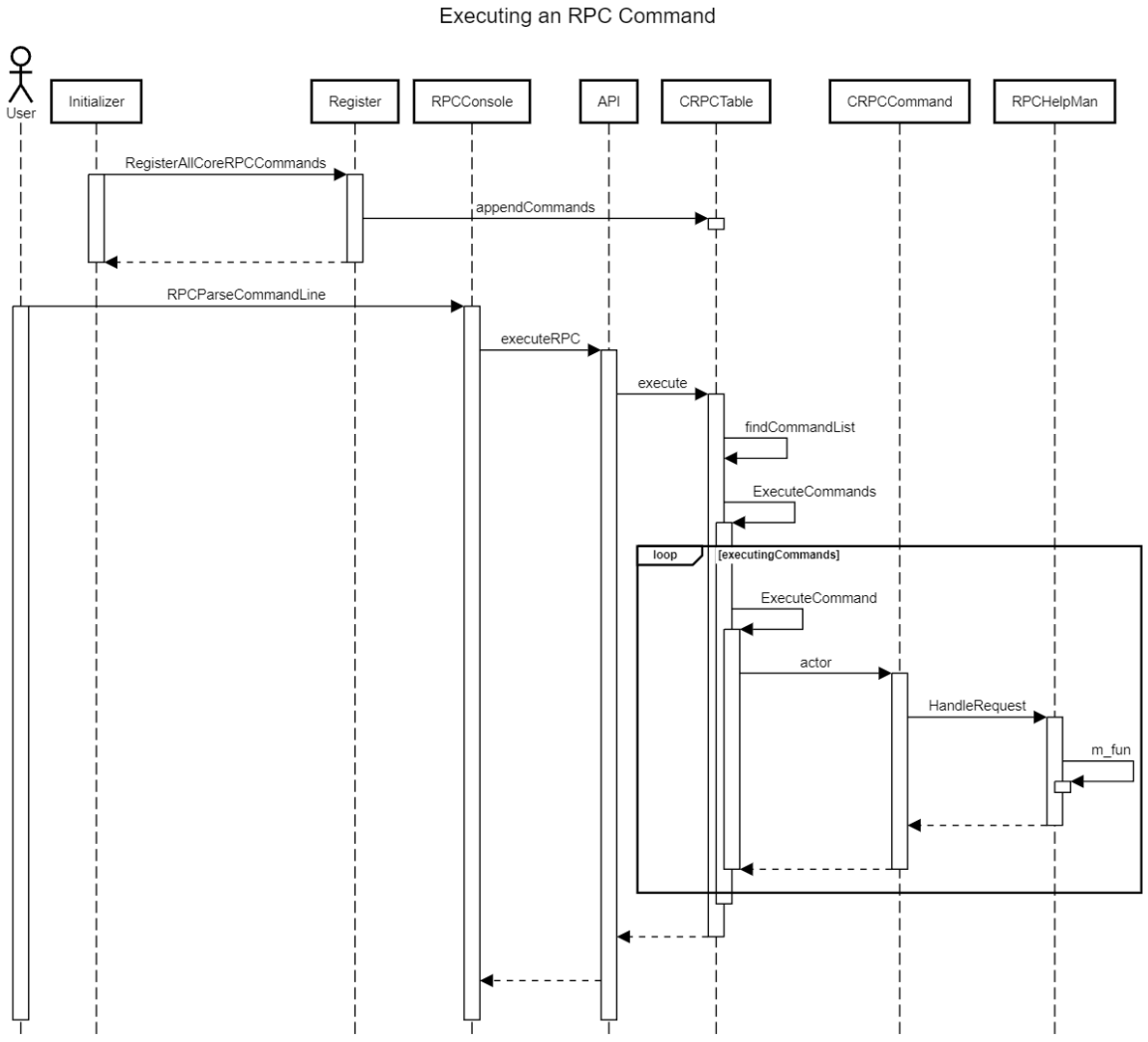

Figure 4: Sequence diagram of executing an RPC command

Use Case #2: Executing the generateblock Command

Assume that everything in use case #1 occurs. The difference is that this use case focuses on the *generateblock* RPC command specifically. Then, *m_fun* that is held by the *RPCHelpMan* object has a pointer to the *generateblock()* method (it actually points to the lambda function inside *generateblock*).

Once *m_fun()* or *generateblock()* is invoked, it first uses the address that was passed in as an argument to decode it and check if the destination address is valid. This decoding and checking happens in *key_io.cpp* which is part of the *Script* subcomponent of *Utilities*.

Then, the method asks *RPCServerUtil* for *Mempool* (rpc/server_util.cpp). This is so it can extract all the transactions that were passed in as console arguments from *Mempool*. It does this by turning the string arguments into a *uint256* hash, and then using that to map the hash to a transaction.

After getting the transactions, it asks *RPCServerUtil* for the *Chainstate Manager*. This is so that *Block Assembler* can create a new block with CreateNewBlock() with the active chainstate data. Normally, CreateNewBlock() fills the block with transactions, however, since the user specified the transactions, CreateNewBlock() only adds the coinbase (or the first) transaction that rewards the miner.

Once the block is created, it adds the transactions manually to the block and then checks its validity using the *Validation Engine* component (validation.cpp).

Since the block is now ready, it invokes GenerateBlock() to start mining. The method increments the nonce and decrements max_tries until a valid proof of work is found using CheckProofOfWork() (pow.cpp) or max_tries reaches zero.

After a valid proof of work is found, the *ProcessNewBlock()* method of *Chainstate Manager* is invoked. This method stores the block to disk, as well as broadcasts the change out to the network.

Lastly, generateblock() returns a UniValue object that has the block hash of the block that was mined.

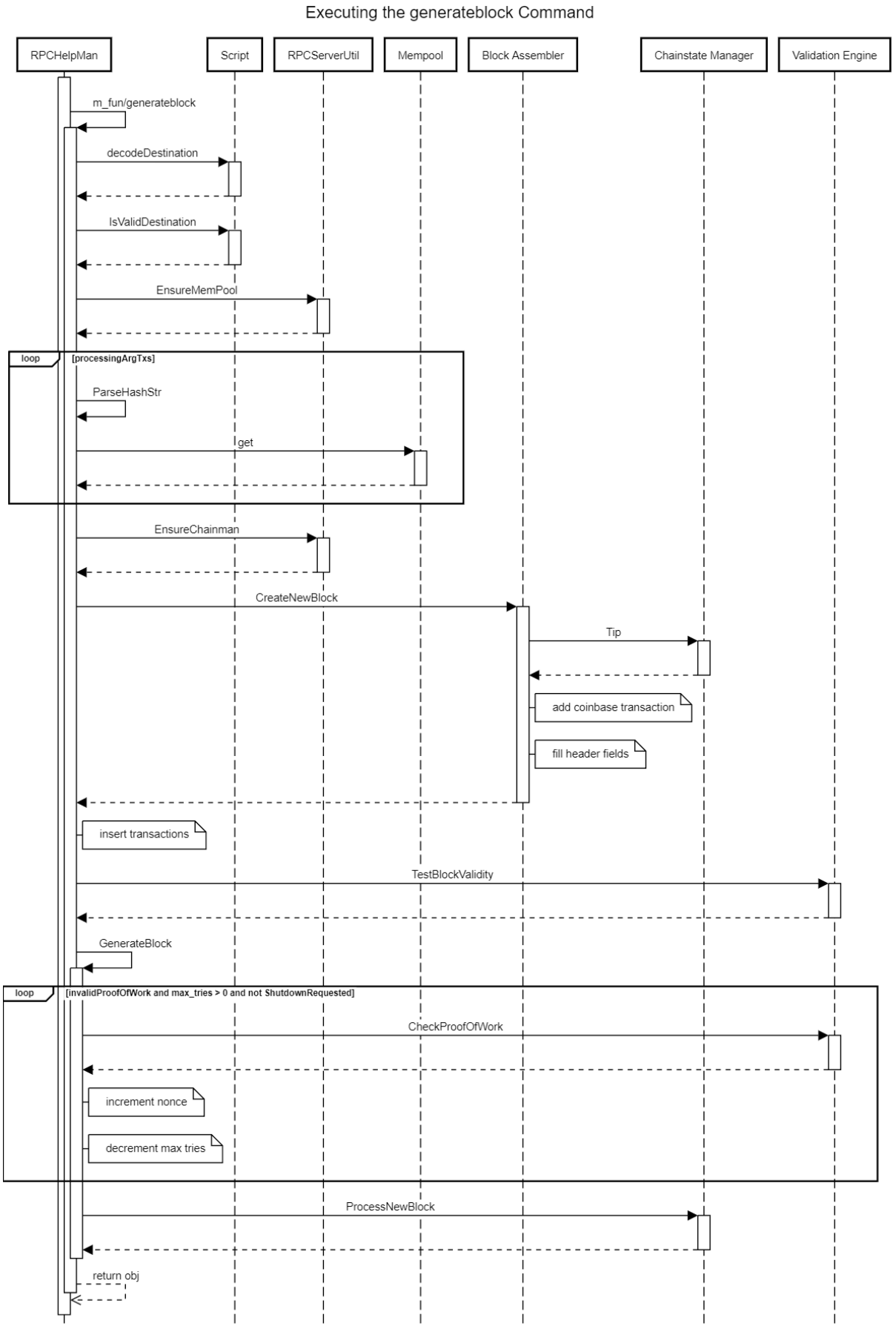

Figure 5: Sequence diagram of the generateblock RPC command

Lessons Learned

Over the course of this assignment we have faced challenges and gained important insights. One major challenge was analyzing the code itself. Without much prior knowledge of the code base, there were instances where we had missed major files that related to various subsystems. Studying the code in the files also proved difficult, and without documentation from the programmers, there would have been even more misunderstandings. We faced this when we tried to do detailed analysis on the *Peer Discovery* subsystem, which proved to be much larger and more complex than we had thought. This ultimately led us to pivot to a simpler subsystem, the *Miner*. The only way we found to alleviate this problem was spending time exploring the dependency graphs and familiarizing with the source code.

Another category of lessons learned is centered on the group and communication. In this assignment, we split our group into two different subgroups, one dealing with the high-level architecture and one with the detailed subsystem analysis. This split affected communication and the two subgroups rarely communicated or shared work. Some team members also didn't communicate that they were too busy to finish their share of work until the day of the group meetings. In the future, we will ensure that members communicate earlier about their situation and also encourage more communication amongst the group.

Naming Convention

RPC - Remote procedure call
API - Application programming interface
P2P - Peer-to-peer

Conclusion

In conclusion, the modified conceptual architecture of Bitcoin Core has been updated to include two new modules, *API* and *Utilities*, while Connection Manager is no longer included. The newly added *API* module provides protocols, routines, and tools to build software applications that interact with the Bitcoin network, while the *Utilities* module provides essential functionality for handling and processing blocks and transactions within the blockchain network. Unexpected dependencies were identified, including bidirectional dependencies between *Peer Discovery* and *Storage Engine*, and *Validation Engine* and *Peer Discovery*.

After analyzing *Miner,* it was also concluded that it is responsible only for the assembly of blocks and unconfirmed transactions into a block template and not mining. Instead, the mining functionality is in *RPC*. As expected, *Miner* interacts with only *Mempool* and *Validation Engine*. However, a divergence was found where *RPC* interacts with *Miner* to assemble the block for mining. An unexpected interaction was found where *Miner* acquired the block subsidy from *Validation Engine*. However, the

block subsidy calculation being inside *Validation Engine* was agreed to be logical since block subsidy relates to consensus and policy.

References

Bitcoin RPC Reference: *RPC API reference*. Bitcoin. (n.d.). Retrieved March 19, 2023, from https://developer.bitcoin.org/reference/rpc/

Peer Discovery: *Bitcoin Core 0.11 (CH 4): P2p network*. Bitcoin Core 0.11 (ch 4): P2P Network - Bitcoin Wiki. Retrieved February 10, 2023, from https://en.bitcoin.it/wiki/Bitcoin_Core_0.11_(ch_4):_P2P_Network

Github Repo: Bitcoin. (n.d.). *Bitcoin/Bitcoin: Bitcoin Core Integration/Staging tree*. GitHub. Retrieved March 19, 2023, from https://github.com/bitcoin/bitcoin