CISC 322 / 326

Concrete Report: Bitcoin Core

Thursday, March 22, 2023

Group 36:

Victor Ghosh (18vg5@queensu.ca)

Ethan Davis (19ead6@queensu.ca)

Stefan D'Ippolito (19sadi@queensu.ca)

David Shen (19ds71@queensu.ca)

Table of Contents

Abstract

The concrete architecture of the Bitcoin Core software implementation has been derived with the intention to compare it to the previously discussed conceptual architecture. The following report will focus on the main subsystems present in the backend of said software architecture. Although the subsystems described in the conceptual architecture are largely accurate there are a number deviations which will be explored. These include multiple unforeseen dependencies and the omissions of some which were expected to be bidirectional. Furthermore, two conceptual subsystems were combined for the convenience and relevance of the concrete architecture.

Introduction

Beginning in 2009, Bitcoin became a revolutionary new form of crypto currency. The idea was to create a form of monetary transaction that did not rely on any central bank or authority to verify purchases. This was achieved through the use of Blockchain technology and incentivized Bitcoin “mining” that would maintain the security of transactions. Thanks to this mining process the current Bitcoin blockchain now contains over 750,000 blocks, each one further securing the block before it. Since its creation the network has expanded significantly and includes a variety of different types of nodes. The following paper intends to deconstruct the Concrete architecture of a reference client node, also known as Bitcoin Core. We will discuss its components, the derivation process, and use cases, while also doing a deep dive into the wallet subsystem.

Top-Level Analysis

Derivation Process

The team began the concrete architecture derivation process by reviewing the current documentation found in the GitHub source code for bitcoin core. Here we could see discussions and planning documents for the code base that gave us a general idea about where the current developers believed components were and where they wanted them to be. We then began reading over the codebase in both source files and the *Understand* software to group the most obvious files into their respective component folders. It was at this point we also designated which portions of the source code we would emit from our analysis such as testing cases and general utility functions. Once a broad map had been formed we read the remaining files more carefully to place them in their most fitting location. Finally, to refine our understanding we focused on code segments which violated our conceptual understanding of the system. This is because it is possible, as was the case on multiple occasions, that we had misplaced a file due to negligence and thus it was causing an unexpected dependency. If this was the case we would move this file and if not we would need to look again at the code and its repository history during our reflexion analysis.

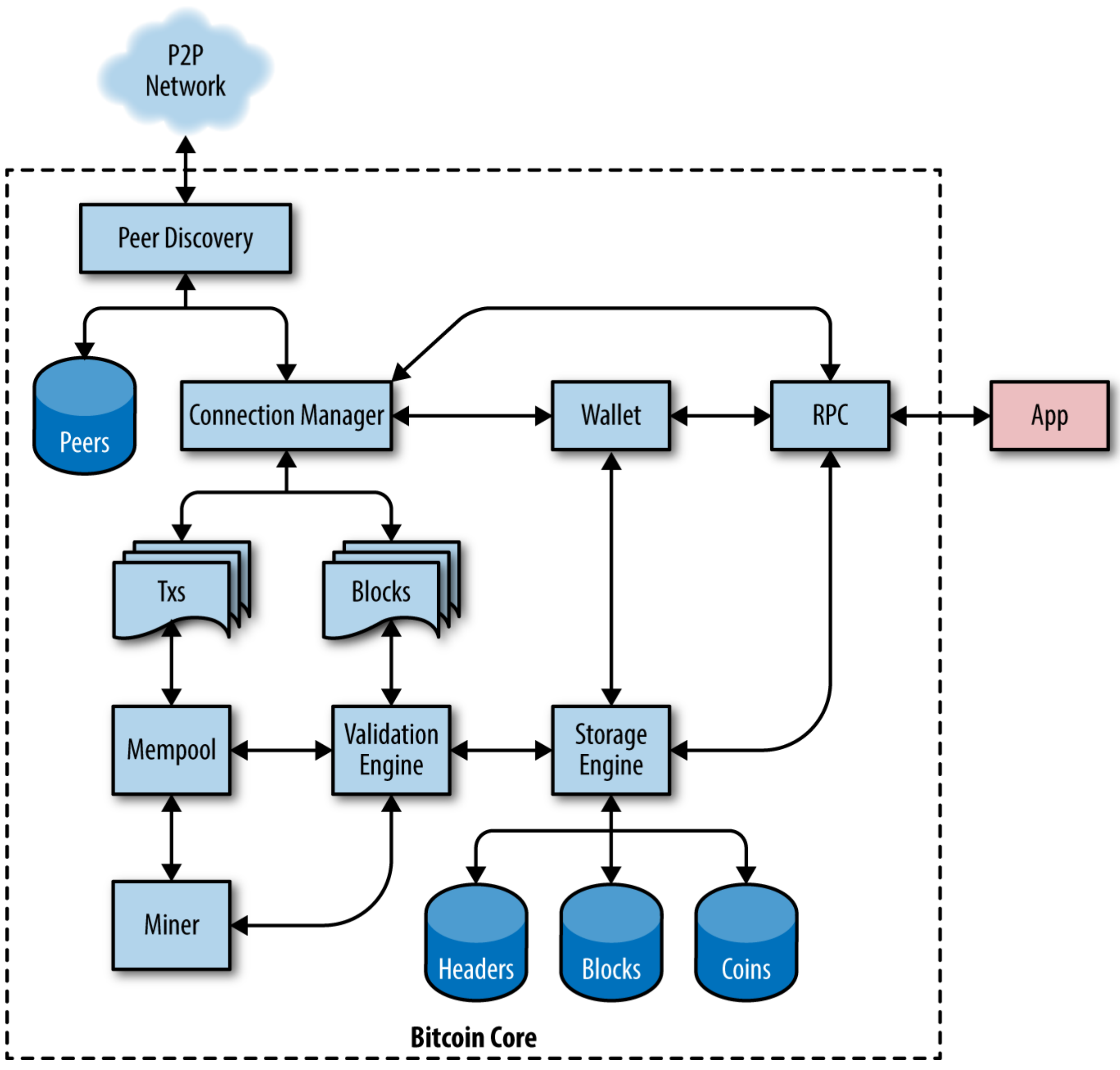

Figure 1: Conceptual Architecture of Bitcoin Core

Concrete Architecture

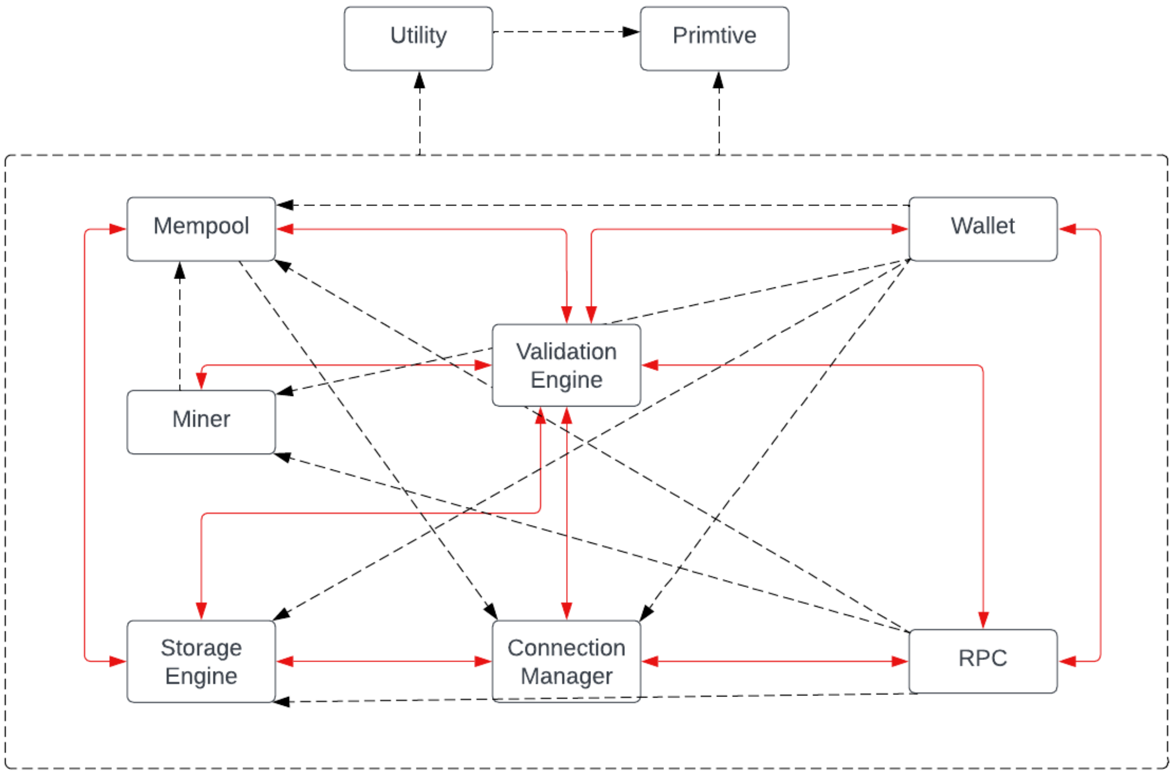

Figure 2: Concrete Architecture of Bitcoin Core

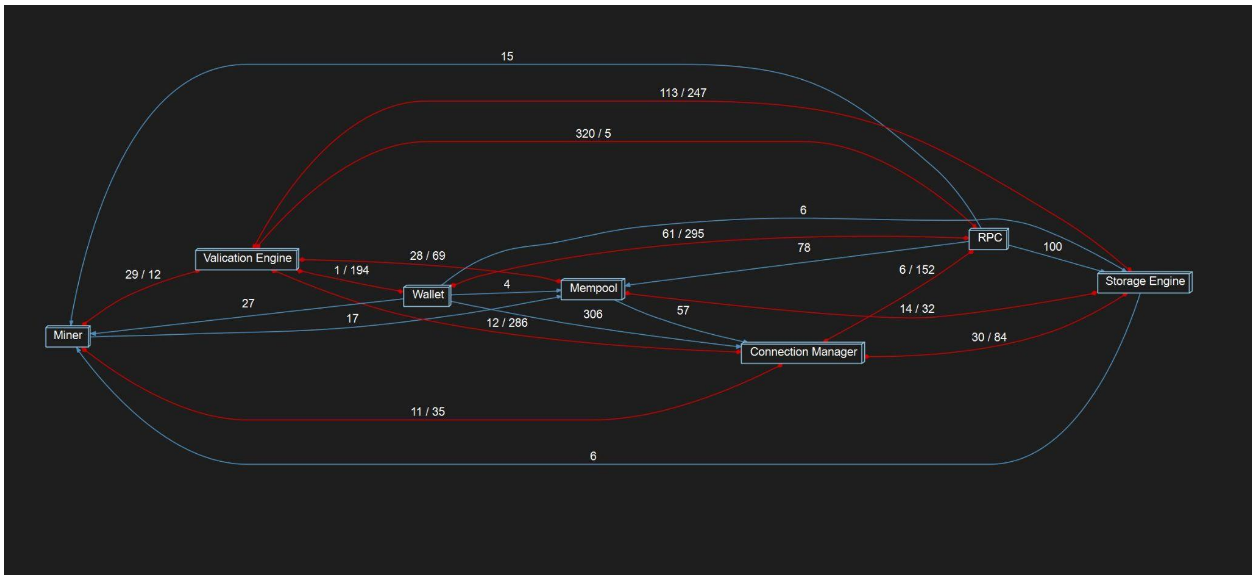

Figure 3: Understand Dependencies Graph

Subsystems and their Interactions

Our group initially discussed eight main subsystems based on the fundamental functions of the bitcoin core software. The majority of this subsystem division is still relevant in the concrete implementation of the bitcoin core architecture with the exception of the peer discovery component. When breaking down the system we found that from a theoretical point of view it made sense to have a separate process to discover new peers and maintain the peer list but, in practice due to all other communication with these peers being handled by the connection manager, there is no true distinction between the two. Therefore, as shown in the concrete architecture the peer discovery has been combined under the connection manager name.

Also in the concrete architecture you will find two additional components: utility and primitive. These refer to a group of classes continuing widely used functions for upper and lower level functionality as well as the main types of data formats the rest of the components rely on. They will not be discussed in depth due to their low impact on the wider architecture

Not displayed in the concrete architecture is the front end interfaces and application functionality of the system. This component as one might have expected, is not one component at all and to include would require an entire report of its own. Needless to say, the front end and qt portion of this implementation of the bitcoin core architecture depend on almost all the substems discussed.

Top-level Reflexion Analysis

By nature of being an open source code base there are no shortage of concrete architecture deviations from the conceptual one. Many of these have been documented as necessary while others have been mentioned in future products for replacement. Some others still remain undocumented.

Wallet -> Mempool

The rationale for this unexpected dependency has to do with the wallet subsystems ability to track transaction/block conflicts. Due to a variety of reasons there may at times be discrepancies between the wallets view of transactions and new ones that the mempool has received from an external node. In order for the wallet to verify future transactions it must rectify this or else there is no way for the wallet to differentiate between conflicting blocks and an unconfirmed transaction. The issue has been documented in the code base and is due to be fixed, likely by updating the wallets heuristics for checking transactions.

Wallet -> Miner

This dependency has to do with the wallet public key hashing using hash functions contained within the miner subsystem.

RPC -> Miner

This new dependency has to do with the RPC needing access to the latest block template for it to create new transactions and send them to the network. Since the Miner has access to the latest block template, without access to the Miner, the RPC would not retrieve the template and the RPC would not be able to estimate transaction fees or create valid transactions.

RPC -> Mempool

This new dependency has to do with when the RPC needs to estimate transaction's fee rates. The RPC searches the Mempool to get information on the state of the market through unconfirmed transactions in the Mempool. The RPC then uses this information to set appropriate fees for new transactions being created.

Subsystem Analysis (Wallet)

Conceptual View

Main Style: Peer to Peer

The Wallet subsystem uses a peer to peer style due to its nature of receiving data and changing the values through a transaction, it also requires the knowledge of the coin itself, other fees, and returns them to an interface. Wallet intakes data from various sources thus, why the subsystem needs to be flexible and have multiple points of data entry and data exit, which happens in the Wallet Info, Interface and RPC, and Database nodes. Some other nodes inside the Peer to Peer network, include Fees: which intakes information about the wallet and transaction, then returns the data with a fee rate included. Next, Coins, which contains the information about the coins, such as values. Wallet Info is the most important node due to its dependencies with every other node and provides information to the other top-level subsystems. Since the Wallet subsystem is responsible for handling most of the requests users can make including transactions, the architecture needs to be flexible to handle many jobs which suits the Peer-to-Peer style well.

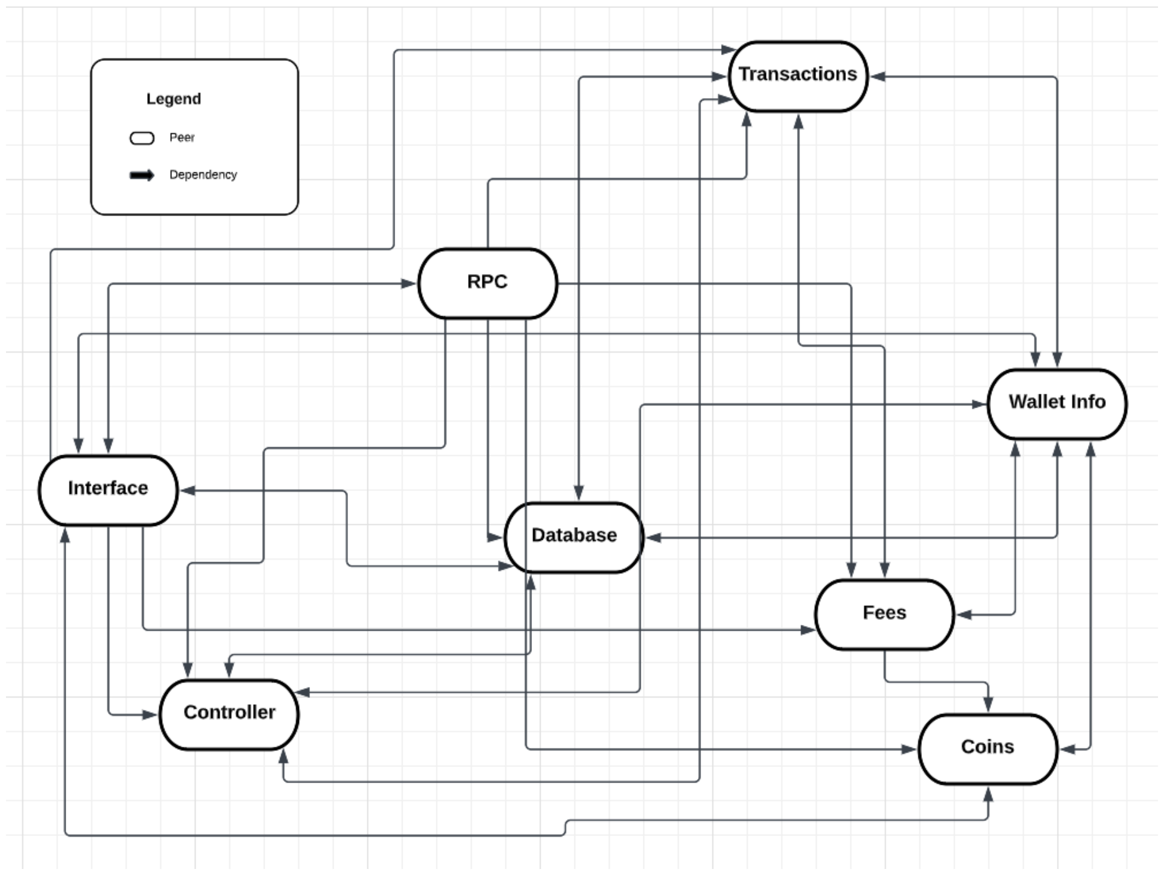

Figure 4 - Conceptual Architecture of Wallet Subsystem

Concrete View

The concrete architecture of the Wallet Subsystem is modeled in *Figure 2.* The Wallet module initially starts with the data intake from three sources, (RPC, Storage Engine, Collection Manager). This data is not gathered at all once but as the wallet module is performing its tasks. The data flow and dependencies can be seen below in *Figure 3* which displays the inner dependencies of the wallet module.

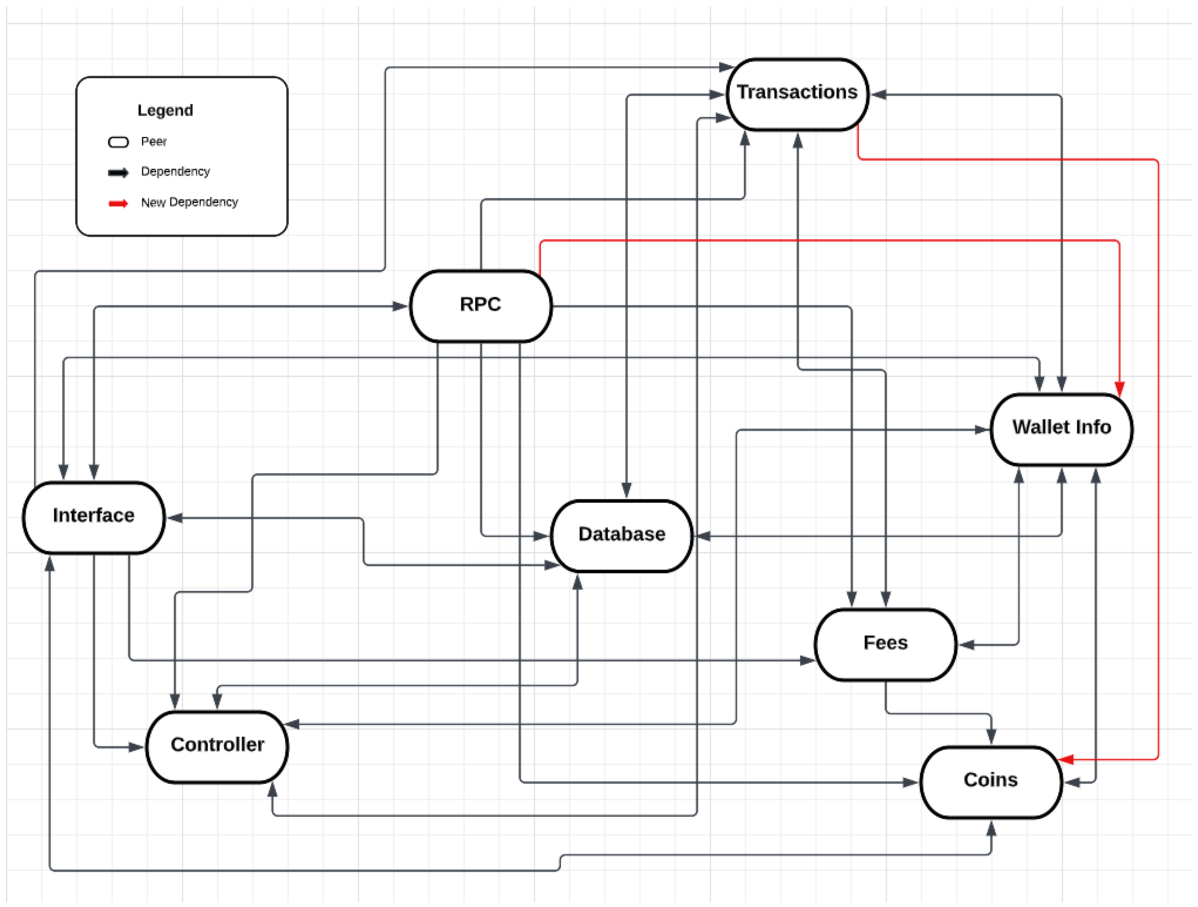

Figure 5 - Concrete Architecture of the Wallet Subsystem

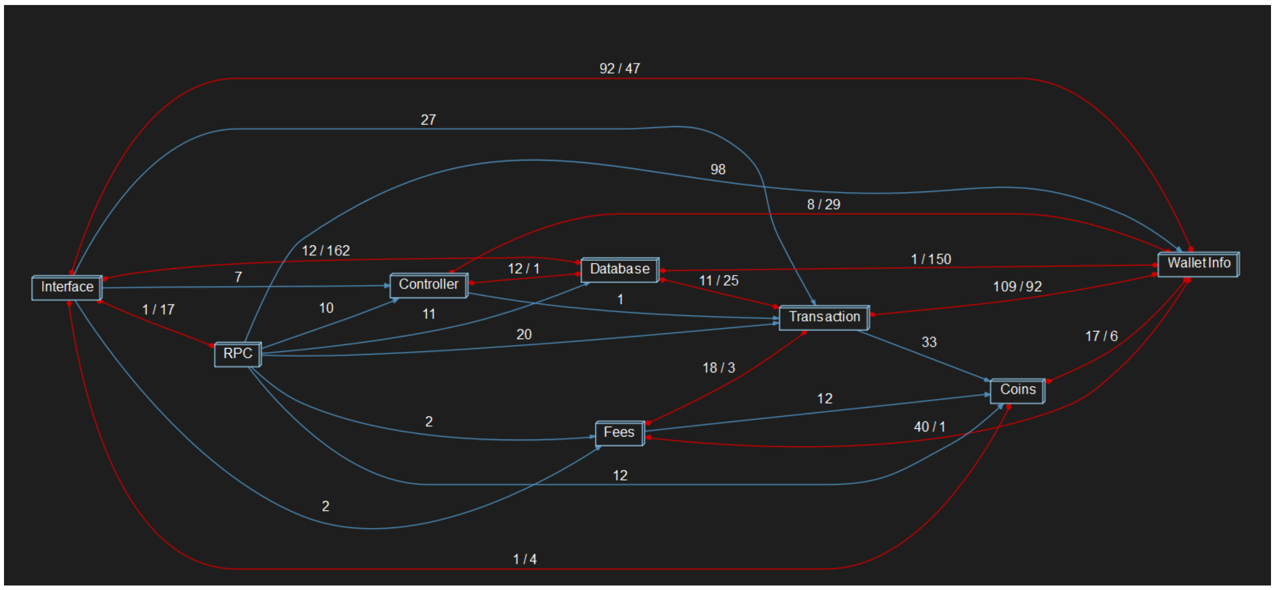

Figure 6 - Understand Dependency Diagram of Wallet Subsystem

The wallet subsystem contains eight essential modules, each interlocking through dependencies to form a peer to peer network. Interface has five dependencies but the largest is the one between itself and the wallet info as it directly sends/receives the data relating to the values and characteristics of the wallet which includes the contents of the wallet, i.e value of the coins. Among the five dependencies of Interface the two way one with the RPC is also crucial to the subsystem as the RPC(remote procedure call) as it helps to disperse some of the data to the smaller nodes in the system. The RPC node is the same as the one seen in the top-level conceptual architecture since at the top level the RPC only has one dependency inside of the source code of Bitcoin Core, and it is the Wallet, in the two figures seen above on the concrete architecture of the specific wallet subsystem, the RPC has many dependencies. RPC only receives data from Interface since its job is to enable programs and scripts to interact with Bitcoin Core, but it does send data to every single other node. This data has been filtered down by the RPC for the other nodes to handle it accordingly. Next, the Controller, which helps manage the processes that the nodes are completing, although this isn’t the case with all nodes in the system. Controller receives data from the Interface, RPC, Database, and Wallet Info, and helps control the processes of the database and wallet info. The Database is the wallet subsystems access to the storage engine, and provides the Controller, Wallet Info, Interface, and Transactions with the data required to complete their processes. The Database node allows these nodes with access to all of the blockchain data, and of every previous transaction. The database also receives data from the same sources after the processes have completed to update itself after with the new data. Fees is a less important node that just handles the fees related to the coins and the transactions, by receiving data from the interface and RPC about the fees. Transactions manages the process of completing, or reversing a transaction, which includes buying/selling of Bitcoin tokens. It depends on most nodes in the system as it requires data of many variables like the price of the token, and address of the token. After the transaction has been completed it sends the new data to Coins, Wallet Info and Database. Coins provides the system a way to track the addresses of the UXTO’s (unspent transaction outputs). It depends on the data from the Fees, Transactions, Wallet Info, RPC, and Interface and sends its post process data including the addresses mentioned before back to the Wallet Info and Interface. Lastly, the Wallet Info is the most important node in the Wallet subsystem. This node provides information about the current state of the wallet, such as the total balance and number of transactions. It shares a two way dependency with every node besides the RPC, because the RPC should only complete its process once a cycle. Overall, the wallet subsystem shares a similar architectural style as the top-level architecture using a peer to peer system, with 8 nodes that share dependencies between them.

Subsystem Reflexion Analysis

When analyzing the conceptual and concrete architecture of the wallet subsystem, there were some unexpected dependencies while performing the reflexion analysis. We will discuss some of the more important unexpected dependencies.

RPC → Wallet Info
RPC or remote procedure control is a module that acts as a communication protocol between nodes in a network. The RPC interface enables programs and scripts to interact with Bitcoin Core, with functionalities ranging from spending electronic wallets to accessing private data. In the subsystem of the Wallet the RPC performs tasks of providing the interface with another method of filtering data and dispersing it. Originally the RPC was thought to only communicate with the nodes that provide jobs for the Wallet Info node, like transactions and fees. However as seen in Figure 3 the RPC sends quite a large amount of data to the Wallet Info. This is because Wallet Info must receive some of the filtered data passed from the Interface to the RPC and then on to the Wallet Info.

Transactions → Coins
Initially, the Coins was thought to function with only information given from the interface, fees and the wallet info modules, because they provide Coins with the necessary info to complete its job which is keeping track of the unspent transaction outputs (UTXOs) associated with a particular Bitcoin address. Transaction sends the data regarding the UXTO which previously was thought to be sent from the wallet info. This is because it is quicker for the Transaction module to send the UXTO information as that is where the transaction occurs.

Use Case: Bitcoin Transaction

The following use cases follow a bitcoin transaction from start to finish. The process of this use case differs only slightly from the one of our conceptual architecture. As stated in the document, there is no Peer Discovery component because Peer Discovery and Connection got merged into the same component. There is also no dependency from Connection Manager to Wallet, Mempool to Miner, and Storage Engine to RPC which were all in our conceptual architecture. So, first Bitcoin Core's Wallet software creates the transaction and specifies the recipient's Bitcoin address and the amount of Bitcoin they want to send. The user's wallet signs off on the transaction using the private key associated with each input to the transaction. This is done to prove that this is the owner of the bitcoin being sent. Once signed off the transaction is created and broadcast across the network through the Connection Manager. The Connection Manager will maintain a small list of known peers to contribute to the network. The Connection Manager will then keep the addresses of these peers in memory and periodically move them to storage. Nodes pick up transactions and check if it is valid through the Validation Engine. If the transaction is valid it is added to the Mempool to wait until it is confirmed by a miner. Once it is confirmed by the miner, the Connection Manager is called and the transaction from the Mempool is sent through the connection manager to the Miner. The Miner takes transactions and recursively hashes pairs of transaction hash values until there is only one remaining, which is called the Merkle Root. This along with timestamp, nBits, nonce and a reference to the previous block hash is placed in the block header. For a transaction to be added to a block in the chain the miner needs to solve a difficult math puzzle based on a cryptographic hash algorithm to show proof that the miner spent significant computing effort (also called proof of work or PoW). After the Miner solves the PoW algorithm the last step is the independent validation of each new block by every node on the network through the Validation Engine. Once it is verified this information is stored in the Storage Engine and is included in a block and recorded on the blockchain.

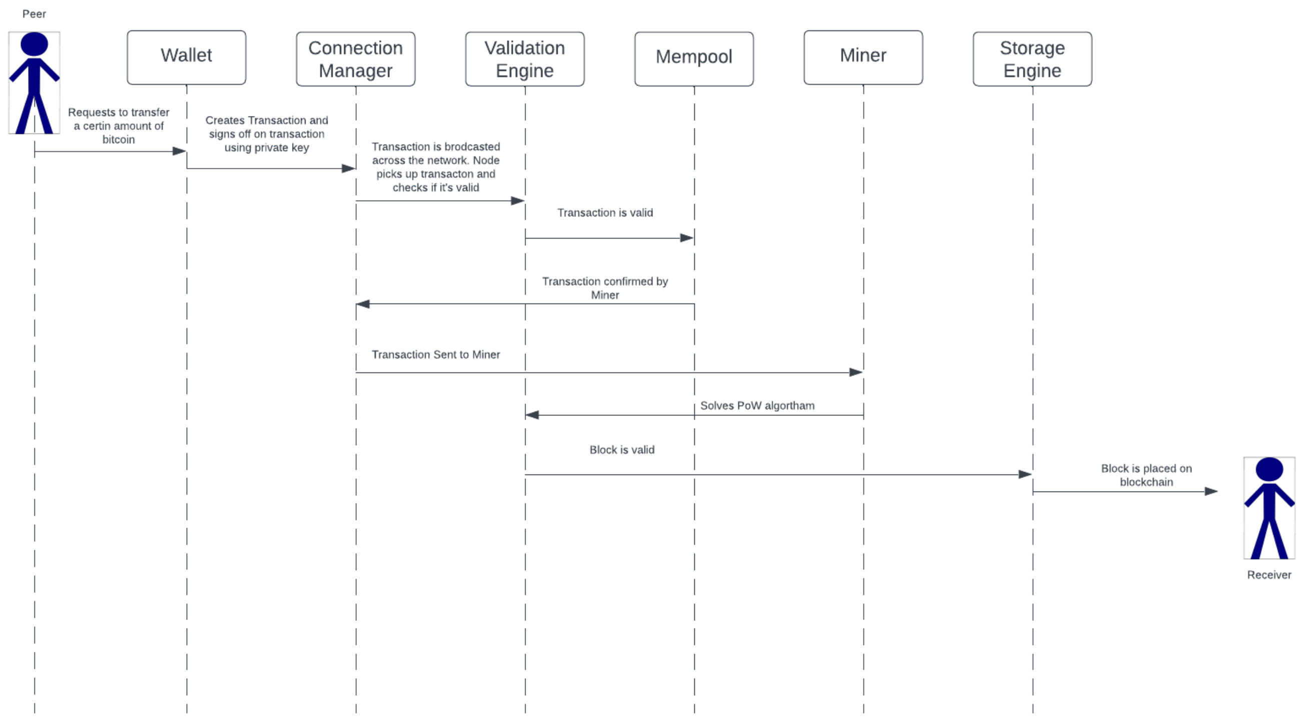
Peer
Wallet
Connection Manager
Validation Engine
Mempool
Miner
Storage Engine
Requests to transfer a certin amount of bitcoin
Creates Transaction and signs off on transaction using private key
Transaction is brodcasted across the network. Node picks up transacton and checks if it's valid
Transaction is valid
Transaction confirmed by Miner
Transaction Sent to Miner
Solves PoW algortham
Block is valid
Block is placed on blockchain
Receiver

Figure: Sequence diagram for Use case 1: Transaction

Use Case Mining

The second use case goes through the process of adding transactions to the bitcoin blockchain. This use case assumes that the node we are viewing is the “winner” of this blocks proof-of-work competition. The node is constantly receiving new transactions from its peer network and retransmitting these transactions to the other neighbors after validating. When a new block is received it validates it and ends its computation on the previous block. It will also slot this new block into its copy of the blockchain. The node then compares the transactions, throwing out any transactions in its memory pool that were included in the new block, before creating the header and constructing the candidate block. The candidate is sent to the miner which will compute the nonce and return the hashed block. It will also send this block to the storage engine so it may be added to the nodes running copy of the block chain and the reward keys can be added to its wallet. Finally, the newfound block is sent to all the node’s peers.

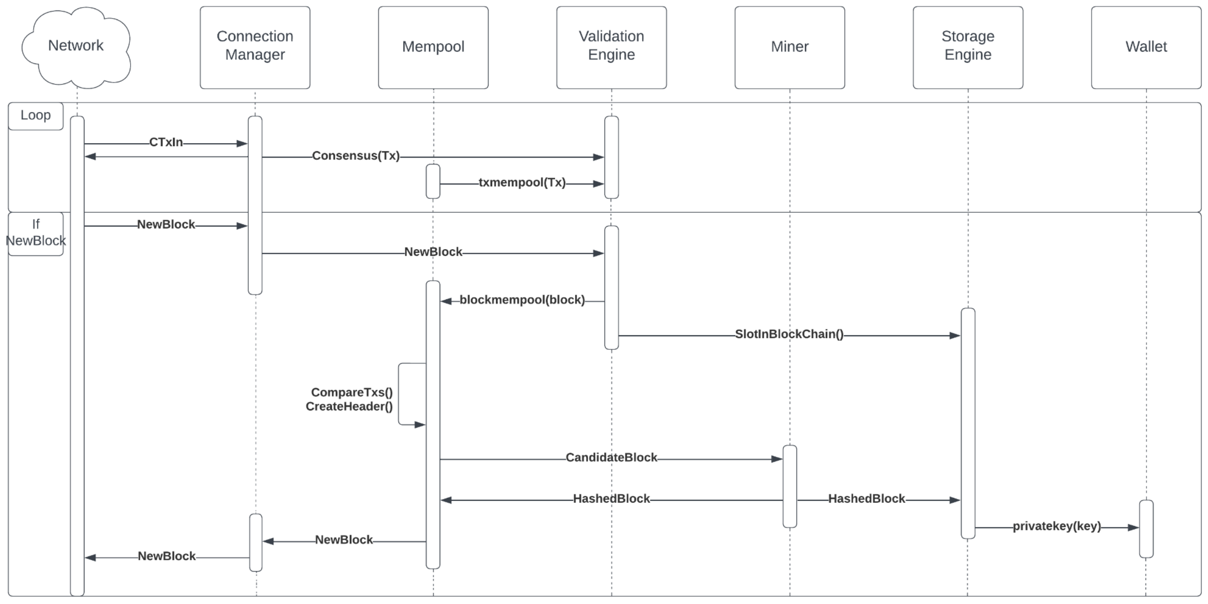

Figure: Sequence diagram for Use case 2: mining

Lessons Learned

A wide range of lessons were learned during the derivation of the Bitcoin Core concrete architecture. The magnitude of this task was not fully appreciated by any of the group members prior to starting but it quickly became apparent that we would not fully understand the code base as we could with the examples found in class. The number of features in the system made this impossible for just the four of us to achieve. Furthermore, many of the features being implemented were beyond our conceptual abilities due to their use of complex algorithms and functions. All these reasons made the importance of architecture design extremely apparent because without ever understanding how each component operates we may create a broad map of how a system functions to execute some larger goal. In addition we can use this high level understanding to communicate to clients and developers what is going on in the code production and maintenance. To some extent this process has shown that even an incomplete or partially incorrect architectural understanding should be vastly preferred to none because it may greatly decrease the time required to implement new features, maintain current features, or accurately estimate future costs.

Conclusion

The concrete architecture of the Bitcoin Core software implementation was derived through the use of *Understand* and git logs and comments found on *GitHub*. The software was segmented into a variety of subsystems which interacted with each other through dependencies. This architecture has been compared to the previously explained conceptual architecture to find any large differences or deviations. In addition to the overall system, a more in depth and slightly lower level examination was done on the wallet subsystem. In this section we explored the components that made up this subsystem as well as the components they depended on. Finally, a reflexion analysis was done on both the high level system and the wallet subsystem. By doing this we found the rationale behind deviations from the conceptual architecture such that we may update our understanding of the conceptual architecture or make suggestions for corrections to the code base.

References

“Bitcoin Core.” *Doxygen Bitcoin Core*, https://doxygen.bitcoincore.org/annotated.html.

---. 24.0.1, 12 Dec. 2022, https://github.com/bitcoin/bitcoin.

“Cypherpunks-Core.” *Mastering Bitcoin*, https://cypherpunks-core.github.io/bitcoinboo