CISC 322/326

Assignment 2: Report

Concrete Architecture of Bitcoin Core

Friday March 24, 2023

Group: All Around Average

Jonathan Sumabat *20js30@queensu.ca*

Lukas Boelling *lukas.boelling@queensu.ca*

Nour Mahmoud *17nhm1@queensu.ca*

Caleb Chiu *20cyjc@queensu.ca*

David Kropinski *19dsk3@queensu.ca*

Jeff Jiachuan Li *19jjl6@queensu.ca*

Table of Contents

1. Abstract

In the previous paper, the conceptual architecture of Bitcoin Core was analyzed on its functionality, interaction, evolution, and control and data flow among its parts. Using the src.und file, our group analyzed the source code to derive the proposed concrete architecture. New subsystems are introduced, more specifically: Utility, Testing, and Cryptography. In this report we analyze the architecture and perform reflexion analysis to learn that our original proposed conceptual architecture is not perfect. The concrete architecture introduces unexpected dependencies and interactions to improve the conceptual architecture.

2. Introduction and Overview

Launched in 2009, Bitcoin core is an open source project which serves as the reference implementation of the Bitcoin protocol. It is responsible for maintaining and updating the Bitcoin network's software. Bitcoin Core is currently being updated and maintained by a community of developers although anyone is able to contribute. It is the most widely used implementation of the Bitcoin protocol and has contributed significantly to the growth, development, and evolution of the Bitcoin network.

Following the first report, which dove into details about the conceptual architecture of the Bitcoin core system, this report will take a look at the system's concrete architecture. Analyzing the source code of the system, we have concluded that the concrete architecture of Bitcoin Core best fits the peer-to-peer style, which was similar to our original conceptual architecture report. In this report, we covered an in depth analysis of subsystems and their interactions with one another. The concrete architecture included all of the conceptual architecture's 8 subsystems: Connection manager module, Peer Discovery module, Wallet module, Storage engine module, Validation engine, RPC module, Mempol, and Miner module. In addition, a number of new and crucial components which handle processing and validation of transactions will be introduced: Cryptographic, Transaction, Testing, Utility, and Block. Each component of the Bitcoin network plays a distinct role and communicates with other components in a peer-to-peer manner.

The network's nodes communicate with one another to disseminate blocks and transactions, confirm transactions, and keep a copy of the blockchain. In order to compete and add new blocks to the blockchain and receive block rewards, miners, who are also nodes in the network, communicate with one another. Without the aid of a central authority or middleman, all of these subsystems communicate with one another directly. This implies that every node in the network is equal and can talk to every other node directly. Increased security, resistance to censorship and government control, and independence from conventional financial institutions are just a few advantages of the decentralized nature of the Bitcoin network. Ultimately, a major element that enables a decentralized, secure, and trustworthy digital money is the peer-to-peer architecture of Bitcoin Core, which is based on subsystem interactions. Of course, this report will go into detail on each subsystem specifically, its purpose and how it interacts with another.

Beyond the concrete architecture derivation and reflexion analysis on the system, our team also reviewed the use cases we had detailed in our initial report to reflect the concrete architecture derived. Finally, we concluded by summarizing our key findings, as well as detailing any limitations and lessons learnt while researching and writing this report.

3. Derivation Process

To derive the concrete architecture of Bitcoin Core, the SciTools Understand application was used to generate a dependency graph to visually examine dependencies of the codebase after looking through the documentation and source code provided. The derivation process started with several members looking through subfolders and individual files to understand the functions and interactions of code. Based on what we believed each file did, they would be grouped up into a specific subsystem to derive the concrete architecture. Once completed, we had several individual files remain, specifically files which we could not find a precise grouping as they encompassed multiple scopes. After some discussion, we decided that the set of files be attributed towards the "utility" module, thus resulting in our final concrete architecture.

4. High-Level Concrete Architecture

Overview

After executing our described derivation process, our group ended up with the high-level architecture seen in Figure 4. We concluded that a Peer-to-Peer style is used, similar to our conceptual architecture. The Peer-to-Peer style was still highly evident due to the large number of dependencies between the Peer Discovery subsystem and other modules. As for subsystems, we for the most part kept all the subsystems from our conceptual architecture and added a couple new systems. We will now take a brief look at each subsystem, viewing its dependencies and functionality within the whole system.

Subsystems

<u>Wallet</u>

The Wallet subsystem is responsible for keeping track of a user's balance, managing a user's private keys and their associated public addresses, along with the transaction linked to those addresses. This data is shared with many other subsystems, such as the Validation Engine, Block, Cryptographic, etc. The Wallet module contains files relating to the storage of private keys and other encrypted data, utility functions for encryption and decryption, creating and signing transactions. The wallet subsystem depends on many other modules, two main ones are the Transactions and Blocks subsystems. They allow the wallet to create and broadcast transactions and to make sure the wallet is updated with the latest state of the blockchain to verify incoming transactions.

Validation Engine

The Validation subsystem functions as a rule enforcer, helping to maintain the integrity of the blockchain by ensuring the validity of transactions and block data. This subsystem will be discussed in further detail later in the paper.

Peer Discovery

The Peer Discovery subsystem is responsible for the discovery and connection to other nodes in the network and enables a node to participate in the peer-to-peer network.

Miner

The miner subsystem is an essential component of the network responsible for creating new blocks and adding them to the blockchain. The miner subsystem ensures each transaction meets network rules and regulations as well as making sure the required fee is included. After verifying transactions, it creates a new block containing a set of validated transactions and other metadata. The process of "mining" a block involves searching for a valid nonce (randomly generated number used to verify transactions or perform security checks), appending it to the hash of a current header. The miner subsystem repeatedly rehasesh the block header and compares it to the target hash until a nonce value results in a hash value that meets the requirements of the target, which a reward is then given.

Storage Engine

The Storage Engine subsystem is used to keep constant up-to-date data about the blockchain, such as block data, chain state, transactions, and set of UTXO (Unspent Transaction Output) on local disk. This subsystem depends closely on the Validation and Block subsystems as before a block is stored on disk.

Mempool

The Mempool subsystem is used to maintain a pool of new and unconfirmed transaction that are received from the network and have not yet been recorded to a block. The Mempool subsystem has files that determine the max size of the mempool, the max percentage of available memory the pool can use, and the minimum fee a transaction must pay to be included in the pool.

Cryptographic

The Cryptographic subsystem contains a collection of cryptographic functions / data structures, mainly hashing.

Block

This subsystem is in charge of overseeing the generation and validation of new blocks.
Miners play an important role in this process, solving mathematical problems that validate transactions and produce new blocks. Newly produced blocks need to be verified by additional network nodes. The validation phase ensures that new blocks are valid, complying with network rules and procedures managed by the block subsystem. This is done through verification, making sure the block's transactions are legitimate and adhere to network regulations. Lastly, the block subsystem manages the reward system incentivising miners to continue creating and verifying network transactions. This subsystem works closely with the Miner, Cryptographic, and Transaction subsystem to ensure the network runs smoothly.

Transaction

The Transaction subsystem's main purpose is to manage the verification, creation, and broadcasting of transactions on the network. This data is shared with other subsystem, such as Block, Validation Engine, etc. The Transactions module contains files which deal with the creation, validation, and processing logic of transactions, and tracking of UTXOs. The Transactions subsystem depends on the Block subsystem to be able to access data regarding the blocks from the blockchain.

Connection Manager

The Connection Manager subsystem has the key responsibility of managing the connections between nodes on the Bitcoin network. The Connection Manager module has files which initialize and process network connection, keep track of a list of peer nodes, and manages connections with other nodes. The Connection Manager subsystem depends on the Mempool subsystem as it manages a list of new transactions from other nodes not yet recorded on blocks that are received by the Connection Manager.

External API

The External API subsystem bears the responsibility of handling command line interface commands, as well as the Bitcoin remote procedure call API. The CLI module of the External API allows users to run commands that can access wallets, manipulate transactions and blocks, as well as access utility functions and more. The RPC API allows users to interact with the blockchain system by querying blockchain-related information, including information from the peer-to-peer network.

Testing

The testing subsystem contains a series of unit tests, integration tests, as well as a module for benchmarks, generated by the bench file based on a method called and its priority level.

Utility

The Utility subsystem contains generalized files that are used by every subsystem. Some examples of this are the logging handler, the random file, and code to prevent deadlocking.

5. Validation Engine (Secondary Subsystem Conceptual and Concrete View)

Conceptual View

The validation engine uses a layered architectural style.

Validation Constants

There is a common constants subsystem which holds constants relevant to validation which are expected to be used across multiple subsystems in the validation engine.

Script and Fees

At the lowest level there are the script and fee subsystems. The script subsystem handles bitcoin's script language which allows transactions to be spent. The fee subsystem works with transaction fees to check if transactions are worth including in blocks for mining, as well as estimating how long transactions will take to be included in a block on the network.

Txn Verify

Above the script and fee subsystems is the transaction verification subsystem. This is responsible for checking the validity of transactions and deciding if they follow the consensus rules of the bitcoin network as well as user/miner preferences. It must check that the scripts contained in transactions are valid and whether transaction fees match user/miner preferences and thus it has dependencies on the script and fee subsystems.

Validation

The top-level validation subsystem provides high-level validation functions which act on blocks and transactions, as well as an interface to subscribe to be notified about validation events whenever they occur. It naturally has a dependency on the transaction verification subsystem.

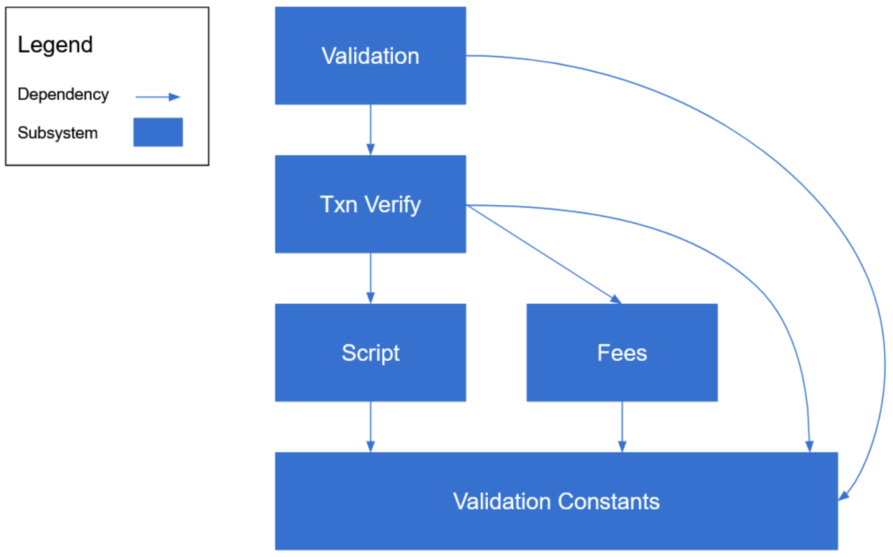

Figure 1, validation engine conceptual architecture

Concrete View

The concrete architecture sees some conceptual subsystems decomposed into multiple more focused subsystems. Still, the same layered style from the conceptual architecture still applies although a few unexpected dependencies have been introduced and are detailed later in the report.

Transaction verification has been split into three primary subsystems:

- Check, which contains transaction checking code which doesn't depend on the blockchain or mempool state.
- Verify, which contains transaction verification code which calls server functions or depends on server state.
- Policy, which enforces preferences/restrictions that are not inherently required by the network or consensus rules.

Utility common to these three subsystems is located in the core subsystem.

The script subsystem has been split into pubkey scripts, signature scripts, and the script interpreter.

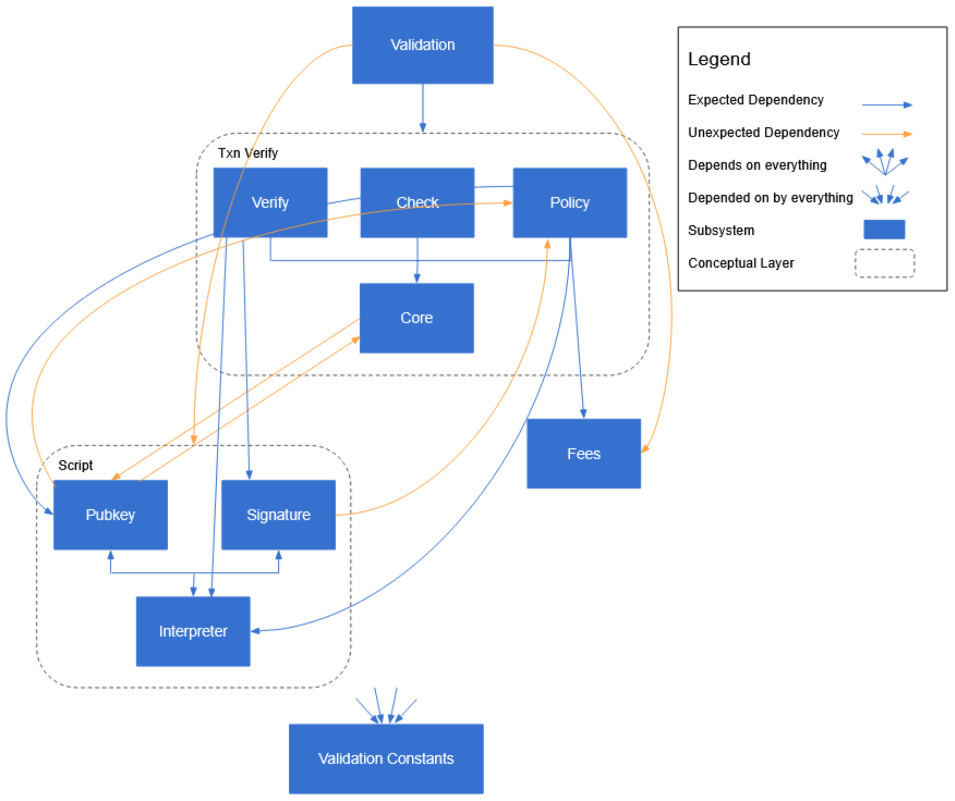

Figure 2, validation engine concrete architecture

Unexpected dependencies

Validation ⇒ Script

The validation subsystem depends on the subsystems in the conceptual script layer, but these are mostly references to constants and types.

Validation ⇒ Fees

When considering whether to accept a package of transactions to the mempool, a check involving the package's feerate is performed.

Pubkey ⇒ Policy

The only dependency here is a single include statement. This seems to be an unused dependency.

Signature ⇒ Policy

The dependencies are a single include statement and a bitfield of standard script verification flags.

Pubkey ⇔ Core

Pubkey uses a function defined in core to calculate a transaction's weight.
Core has a couple dependencies on pubkey for working with the Segregated Witness protocol upgrade.

6. High-Level Discrepancies

Overview

The Bitcoin core architecture is much more interconnected than the conceptual architecture shows. The conceptual architecture mostly shows the flow of data in the system, with each module having few connections. However, when looking at the dependencies each module is shown to be connected to more modules in the system than it is not connected to. This is largely due to many modules having common functions in either processing data or basic sanity checks of data. Furthermore with the introduction of the new subsystems they all have many new dependencies. Figure X below shows the conceptual architecture and figure X+1 shows the concrete architecture that was derived using the understand tool. For clarity the "applies to all" arrow in figure X+1 denotes there being dependencies between it and all other modules. This does not necessarily mean 2 way dependencies with all, more explanation for each of those modules will be given in associated paragraphs.

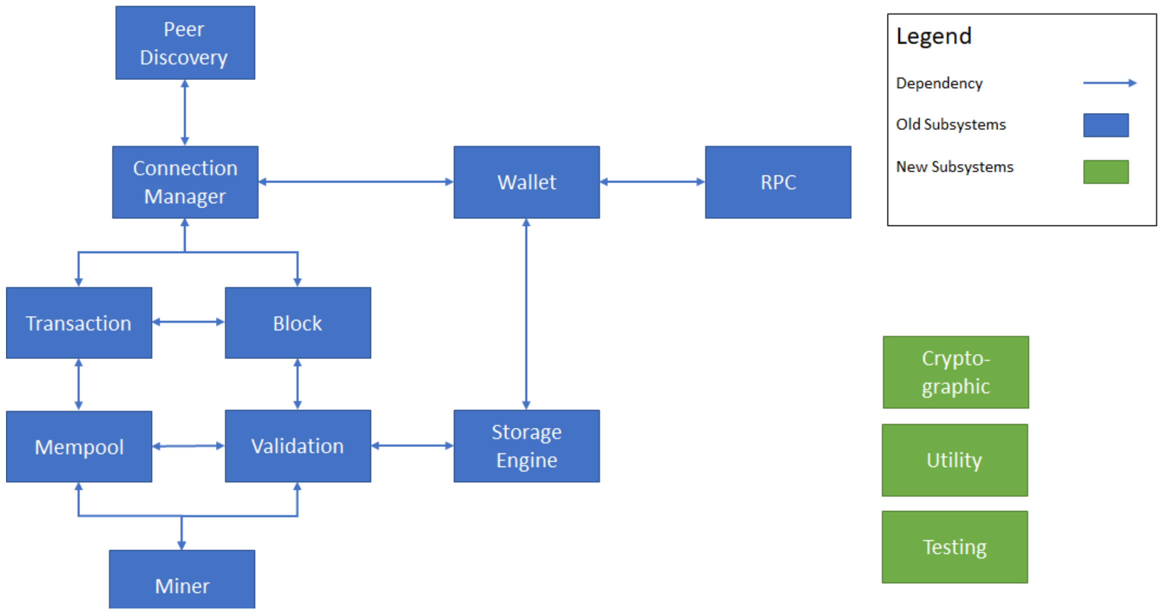

Figure 3, conceptual architecture of Bitcoin Core

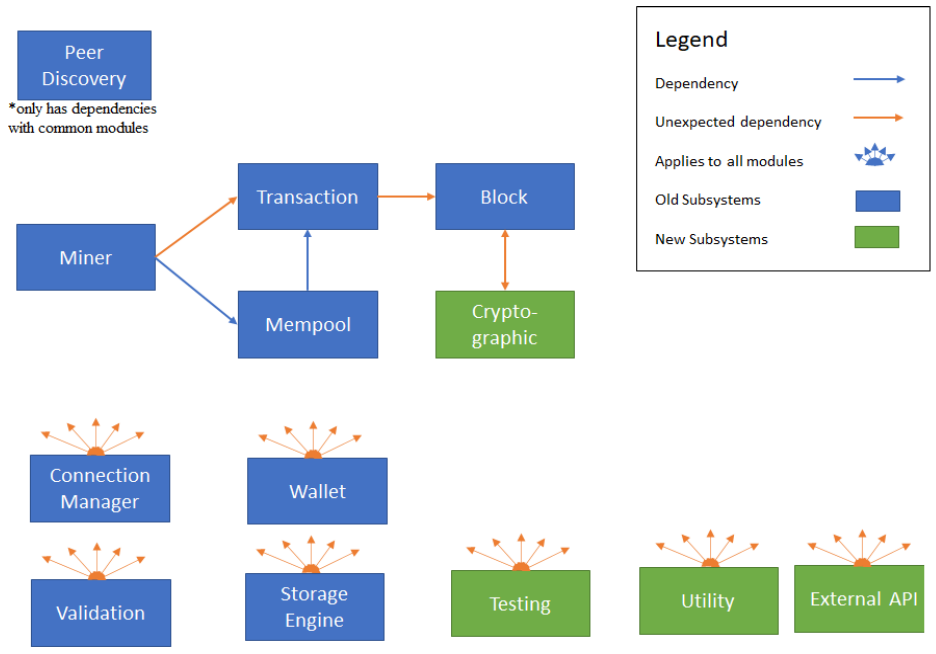

Figure 4, concrete architecture of Bitcoin Core

Utility ⇔ All

Since this subsystem was not in the conceptual architecture all of its dependencies and what depends on it are unexpected. It was not included in the conceptual architecture as it does not change the overall data flow or functionality of the system, its just for efficiency.This module contains many common functions that are used by the other modules such as interfacing, running commands/processes, arithmetic, error handling, logging, and compatibility. The utility module itself does depend on some of the other modules. The main three it depends on are the cryptographic, validation, and script modules. The utility module uses the encryption methods from the cryptographic module, consensus rules from the validation module, and the script module for the handling of scripts in transaction methods.

Testing ⇔ All

This module also was not present in the conceptual architecture. It was most likely excluded as it is mostly used for the testing of the code itself and not used in the actual running of the system. It depends on all the other modules so that it can test their methods. There is one method that relies on testing which is the wallet module. This appears to be because the wallet can run the wallet related tests on itself.

External API ⇔ All

This module was partially present in the conceptual architecture as the RPC module. The external API is a grouping of the RPC and the command line interface. The conceptual architecture shows the RPC as just interfacing with the wallet but in actual fact it can control much more of the system than that. The other reason it has much more connections in the concrete architecture is because of the inclusion of the CLI. The CLI has many functions that allow the user to change/affect the other modules in the system. This module is greatly still depended on by the wallet module. It is also depended on by the connection manager as they share some network related functions and methods.

Storage Engine ⇔ All

The storage engine module in the conceptual architecture was just connected to validation and wallet. However in reality it is depended on by almost every module as it contains methods for handling blocks and the blockchain. These methods are used to get data out of these objects. Since most of the modules work with these types of information they rely on the methods contained in the storage engine. The storage engine itself depends mostly on transaction, utility, and validation. It takes transaction handling methods from transactions, lots of common functions from utility, and some basic block and transaction checks from validation.

Validation Engine ⇔ All

The validation module is depended on by almost all the modules. This is due to all modules doing some amount of validation and testing of data. The conceptual architecture does not show a connection between the validation module and the network related modules, but in fact these modules do run some checks on incoming data. The validation module like the others depends on lots of the data handling methods present in other modules. It also takes some cryptographic related methods from the cryptographic and miner modules.

Connection Manager ⇔ All

Like the previous modules the connection manager relies greatly on data management methods and validation checks that are contained in the other modules. It also has a large dependency on the logging functions in the utility method. It does not have many modules that depend on it. The main one is testing because testing has a lot of tests related to network connections and incoming data. The main mutual dependency is with the external API as they both share some network related functions.

Wallet ⇔ All

The wallet module mostly depends on other modules. Its major dependencies are on the cryptographic module for its many key related functions, the storage engine for its block and chain handling functions, validation engine for the validation of transactions and signing rules, and the utility module for its many common functions. It also has some minor dependencies on the other modules for a couple basic data reading functions. The only module besides testing and utility that depend on the wallet is the external API module that includes some wallet interfacing functions The unexpected dependencies between wallet and the other modules are due to the use of common functions.

Transaction ⇒ Block

This connection is only present because the blocks module contains some of the Bitcoin version information which is important for handling the coins in the transactions. This was most likely excluded from the conceptual architecture as they don't actually share any functionality, it is just so there are no duplicate constants.

Block ⇔ Cryptographic

This dependency was not in the conceptual architecture as the cryptographic module was not present. This is a mutual dependency with each of the modules using hash related functions that are in the other module.

Miner ⇒ Transaction

The miner has one dependency from the transaction module which is the coin.h file that looks at unspent transaction information. This discrepancy from the conceptual architecture was probably not included as it is just a bypass around using the mempool.

7. Use Cases

The sequence diagrams illustrated represent the use cases chosen for the A1 report but reflect the concrete architecture derived in this report.

Use case 1: Initiating a transaction

In this sequence diagram, the sender initiates a transaction by using the concrete method MakeWalletTxOut() to enter the amount of bitcoin and the receiver's address. The sender wallet requests from the transaction module to generate a new transaction that has the bitcoin amount and the receiver's address. The transaction module then checks with the block module to identify the sender's private keys. The Block module then checks the keys with the Cryptographic module. Then, the Block module checks if the sender has the appropriate unspent transaction outputs that represent the input amount. If the sender has enough unspent transaction outputs, the sender wallet requests a digital signature (i.e private keys) from the sender to prove ownership of the bitcoins being sent. The signed transaction is then validated by the validation engine then broadcasted to the Bitcoin network. If the sender does not have the appropriate amount, the sender wallet displays a transaction error.

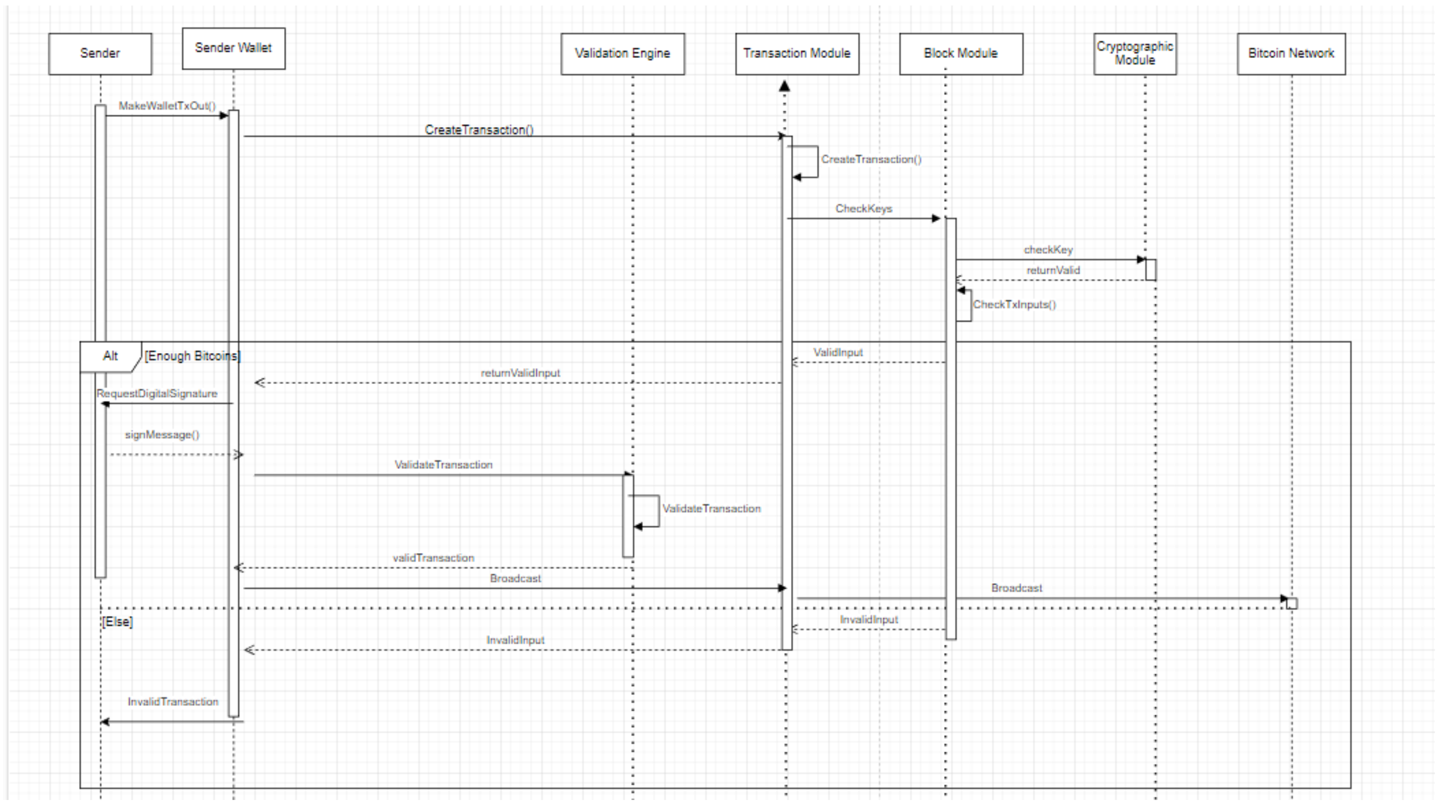

Figure 5, sequence diagram of a user initiating a transaction

Use case 2: Processing a Transaction

This graph shows the interactions between components involved in processing a Bitcoin transaction. The sender initiates a transaction by using the sender wallet to enter the amount of bitcoin and the receiver's address. The sender wallet requests a digital signature from the sender to prove ownership of the bitcoins being sent. The signed transaction is then broadcasted to the Bitcoin network. If the transaction is deemed valid by the network, the Block module requests from the miner to add it to a block of unconfirmed transactions after using the Proof of Work (PoW) algorithm to solve a mathematical puzzle. The block is then broadcasted to the validation engine, which in turn uses the Consensus module to check it against the consensus rules of the network and then it gets added to the Blockchain.

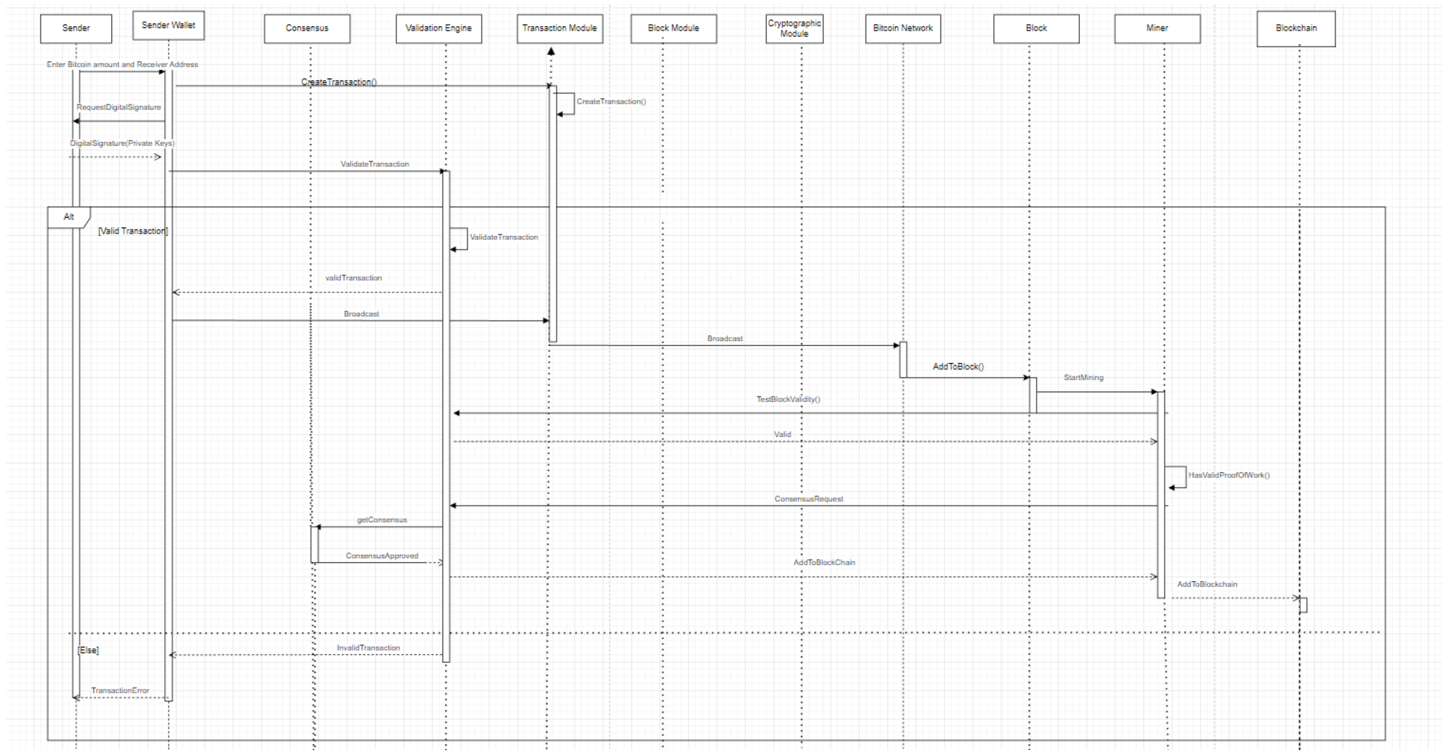

Figure 6, sequence diagram of processing an incoming transaction

8. Lessons Learned and Limitations

Bitcoin core's architecture is far from simple and is built up by a series of complex subsystems and components which work together to ensure transactions run smoothly. It was the first time any of us had used the Understand software and as working with any new technology, there is bound to be some learning curve with understanding how it operates. There were some technical issues at the beginning with setting up the software. However, once that issue was resolved, we were able to use the software to group files into abstract subsystems to find and investigate dependencies between subsystems.

Through investigation and research into the architecture of Bitcoin Core, we quickly realized the importance of thorough communication within our team. This was especially prominent when it came down to organization and distributing work amongst each other. We found a lot of success working together and discussing collectively as a team rather than having each individual derive their own concrete architecture themselves. This ensured two things: One, if there were any questions or calcification needed, group members would have the support available and second, saved a lot of time since we discussed and formulated our ideas collectively which made sure everyone was on the same track. This increased productivity and saved us a lot of time while allowing us to develop a thorough understanding of the Bitcoin Core's architecture. For the future, we plan on continuing to hold regular meetings which allows us to communicate with each other, delegate tasks effectively, and make sure we meet deadlines.

9. Conclusions

In conclusion, through our derivation process of utilizing both Scitools Understand and analyzing the GitHub repository, our group determined the concrete architecture of Bitcoin Core's software was built upon 13 subsystems, which resulted in a Peer-to-Peer style architecture. As a team, we dove deeper into the different subsystems and evaluated how they functioned and related to one another. Additionally, In this report, we examined the architecture and performed a reflexion analysis to discover that the conceptual architecture we first proposed was not perfect. Unexpected relationships and interactions are introduced by the concrete design, which enhances the conceptual architecture. Lastly, with this process giving us a better understanding of the Bitcoin Core architecture, we incorporated two use cases to help illustrate the subsystem interactions focusing on the two main processes of initiating and processing a transaction. Overall our team enjoyed working with each other and found success in breaking down the code of Bitcoin Core, analyzing it and deriving the concrete architecture.

10. Data Dictionary

API: Application Programming Interface
CLI: Command Line Interface
RPC: Remote Procedure Call
Txn: Transaction

11. References

[1] Bitcoin Core. (2016). bitcoin/bitcoin/tree/master/src. *GitHub.* Retrieved March 10, 2023, from https://github.com/bitcoin/bitcoin/tree/master/src

[2] Frankenfield, J. (2022, October 24). *Nonce: What it means and how it's used in Blockchain*. Investopedia. Retrieved March 24, 2023, from https://www.investopedia.com/terms/n/nonce.asp#:~:text=The%20nonce%20is%20used%20to,this%20to%20the%20target%20hash.