Concrete Architecture Analysis of Bitcoin Core

CISC 322
Friday, March 24th 2023

Group 39: Yash
Mercy Doan (mercy.doan@queensu.ca)
Yash Patel (19yp29@queensu.ca)
Cain Susko (20cjbs@queensu.ca)
Alice Slabosz (19aas15@queensu.ca)
Shauna Tuinstra (19st58@queensu.ca)
Samuel Lownie (19syl@queensu.ca)

Abstract

In the previous paper we discussed the conceptual architecture of Bitcoin Core. We wrote about our research on components including Nodes, Wallets, RPC, Transactions, UI, Blockchain, Blocks, Validation Engine, and Miners. After taking another look at the conceptual architecture and using the Understand software to research the source code, we've crafted this technique report to thoroughly explain the concrete architecture of Bitcoin Core. In this report, we discuss three new subsystems: Peer Manager, Mempool, and Utilities. In addition to the new subsystems this report details our new dependency findings involving the new and old subsystems. We conclude that the previous conceptual architecture from our first paper is imperfect, and therefore we analyze the unexpected dependencies and interactions to improve our previous research and develop a better understanding of Bitcoin Core. Later on, we delve deeper into the interactions and functions of the Bitcoin Core architecture, including a subsystem analysis of the Blockchain Storage system, and two reflexion analyses for the top-level subsystem and the high level architecture. This report ends with a data dictionary, the naming conventions, and the lessons we learned while conducting our research.

Introduction and Overview

This report aims to deliver an extensive examination of Bitcoin core's concrete architecture, a revolutionary software system facilitating a secure, decentralized digital payment platform. Utilizing Scitools' *Understand* software, our team meticulously analyzed the Bitcoin Core source code to construct a concrete architectural representation derived from the observed dependencies. This investigation empowers us to gain a deeper comprehension of Bitcoin Core's design, execution, and the crucial characteristics and interdependencies among its components.

This report is organized into several sections, beginning with a description of the top-level concrete subsystems, and their interactions. We then detail the architecture derivation process and provide explanations for the inner architecture of the blockchain storage subsystem. Our team utilized a method which is called 'reflexion analysis'. This method entails analysis and rationalizing existing and unexpected dependencies within Understand, using what was taught in class. We analyze both the high-level architecture and the chosen second-level subsystem (storage of the blockchain and other data), discussing the discrepancies between the conceptual and concrete architectures, and providing rationales for the differences. Lastly, the report will include two sequence diagrams for non-trivial use cases, demonstrating how the diagrams match the concrete architecture.

Our analysis reveals that the concrete architecture of Bitcoin Core retains the peer-to-peer style from the conceptual architecture, but diverges in its use of an object-oriented architectural style. Our team found that the dependency graph of the software is complex, with components depending on almost every other component, fitting the object-oriented style. Additionally, the majority of the code is dedicated to the internal structure of a single peer in the network, emphasizing the external P2P architecture style in fulfilling Bitcoin's mission.

In our investigation of the blockchain storage subsystem, we compared the conceptual and concrete architectures, identifying discrepancies through use of the aforementioned reflexion analysis. Our findings showed that the concrete architecture relies on a data directory containing files and folders such as blocks, chainstate, and wallet.dat, which play essential roles in the storage and management of the blockchain and other data. This information provides quite valuable insights into the design and implementation of Bitcoin Core's software architecture. Using Understand, we have been enabled to refine our initial understanding on the conceptual architecture, and produce a more precise representation of both, the architecture, and the illustration of these architectures.

In conclusion, this report offers a thorough examination of Bitcoin Core's concrete architecture and its implications on the overall understanding of the software system. By analyzing the main components, their interactions, and the chosen subsystem, our team provides a solid foundation for future research and development for the field of decentralized electronic payment systems.

Derivation Process

Our team visualized the static code dependencies using SciTools' Understand software. We developed a comprehensive representation of Bitcoin Core's concrete architecture based on the emerging dependencies. Initially, we established entities for each subsystem within Understand architecture builder and then mapped the source code into their respective subsystems. The chosen entities for our primary subsystem were determined by all entities which would interact with the key component in the network. As we completed the mapping, we generated an internal dependencies graph and an architecture graph to visualize the dependencies and the structure of each module within Bitcoin Core. We leveraged the gathered data to refile the conceptual diagram from our previous assignment and develop a concrete diagram.

After reviewing Bitcoin Core, our team concluded that the architectural style that was employed was a combination of peer-to-peer and object-oriented styles. This is opposed to our initial proposition of Bitcoin Core utilizing the layered style. We arrived at this conclusion by closely examining our recently developed concrete architecture diagram and delving into the source code. We discovered that the primary components were interacting and exchanging information with one-another, which proves that Bitcoin Core uses the object-oriented style.

High Level Architecture

Through our examination of the Bitcoin Core source code, aided by Scitools' Understand software, we were able to derive a concrete architecture for the Bitcoin Core system. The derived concrete architecture retained the peer-to-peer style from the conceptual architecture we proposed in our first report. This was not surprising as Bitcoin's main purpose is to provide a decentralized system for digital currency exchange and the peer-to-peer architectural style fits well with that purpose. The concrete architecture differed from our proposed conceptual architecture in its use of object-oriented architectural style.

Initially, we had proposed in our conceptual architecture that Bitcoin Core combined the peer-to-peer style with the layered style. In examining the concrete architecture, the layered style we had proposed does not actually seem to fit the system. Instead, Bitcoin Core uses a combination of the peer-to-peer style and the object-oriented style. In our initial research, we were focused on how the different components of Bitcoin Core work together to store the blockchain, create and validate transactions, add new blocks to the blockchain, and maintain consensus in order to provide a secure electronic payment system without the need for trusted third parties.

Upon examining the source code, we found that the main components of Bitcoin Core interact with each other in ways that exhibit the object-oriented architectural style. In examining the source code and using Scitools' Understand software, there were many unexpected dependencies in the concrete architecture. The dependency graph of the software was very complicated and involved components depending on nearly every other component. These tangled dependencies definitely fit with the object-oriented style. In the source code, the object-oriented style seems to be the main architectural style. While the peer-to-peer architectural style is a core element in fulfilling Bitcoin's mission, the majority of the code is dedicated to the internal structure of one peer in the network.

The concrete architecture that we derived from the source code included the following subsystems that were also present in our proposed conceptual architecture: peer manager, node, validation engine, wallet, mempool, blockchain, RPC, GUI. Additionally, the concrete architecture contained two primitive subsystems to represent core objects for the bitcoin software, a transaction subsystem and a block subsystem. After organizing the source code into these subsystems, we found that there were still a large number of files and folders that remained difficult to determine which subsystem they mapped to. The file and folder names were unhelpful so we looked into the code itself to determine what each file's main purpose was. We found that these files and the functions they provided were used greatly by other subsystems, and often provided generic helper functions that all other subsystems utilized. So, we created a new subsystem in our concrete architecture called utility, which contains these files and folders of common utility functions.

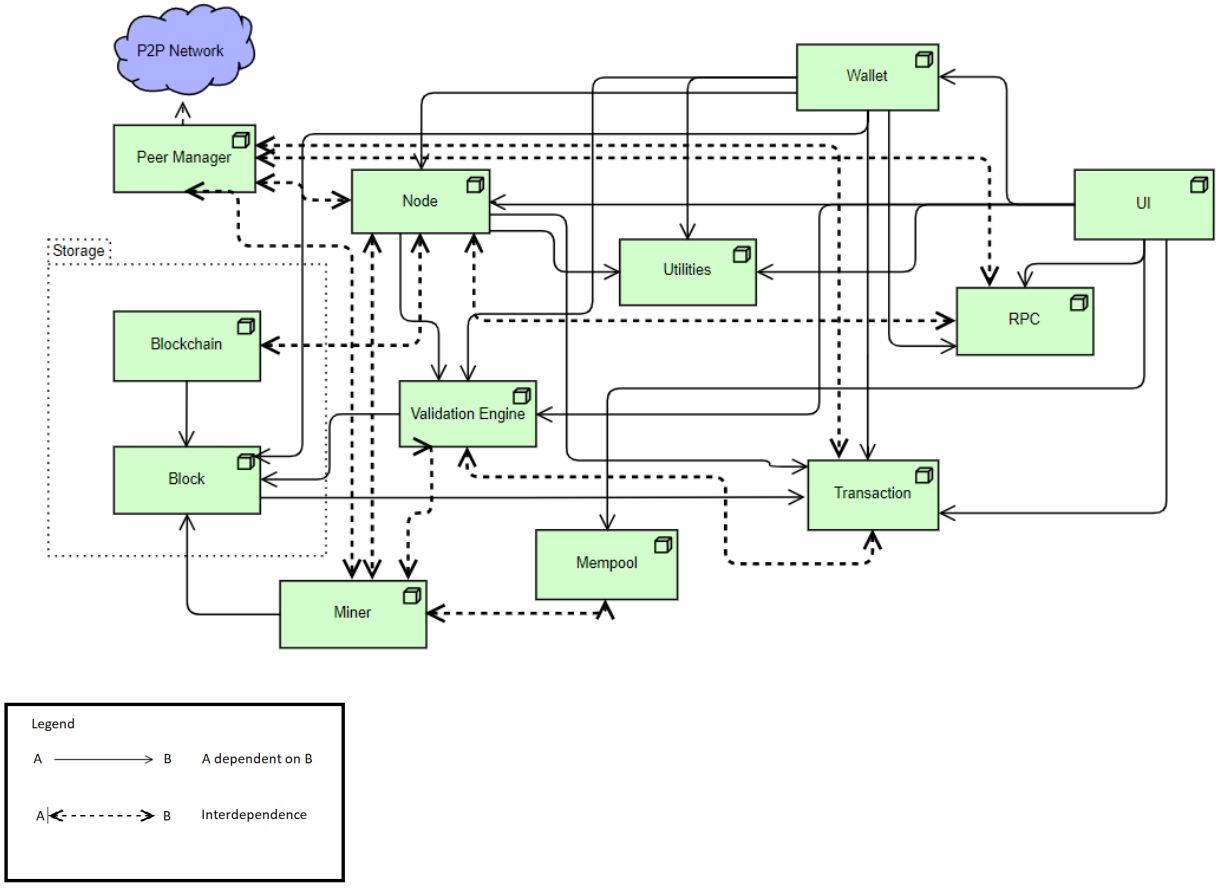

Figure 1. Derived concrete architecture of Bitcoin Core

Concrete Architecture

Table 1. List of subsystems and their responsibilities

Subsystem	Responsibilities
UI	Provides a graphical and command line interface for users to interact with their wallets, create transactions, view their balance, etc.
RPC	The Remote Procedure Call (RPC) subsystem handles sending commands to remote servers and receiving any feedback.
Wallet	The wallet component controls access to a user's money. At a programming level, the wallet is a data structure that manages a user's keys and addresses.
Node	Since Bitcoin Core is built to be a peer-to-peer network, each system running the software must act as a node in the network and must contribute to providing network services. The node component handles these network responsibilities.
Peer Manager	The peer manager component handles the node's communication with other nodes in the network to update its copy of the blockchain when needed, broadcast new transactions and blocks, and participate in maintaining consensus.
Utilities	This subsystem does not have one single responsibility or purpose but rather it is a collection of files/functions that provide common utility functions such as serialization, deserialization, and system settings.
Validation Engine	The validation engine is used to validate transactions and blocks by checking any new proposals against its stored rules for valid transactions.
Transaction	A primitive subsystem that provides the basic definition, formatting, and creation of a transaction. A transaction is a data structure that encodes the transfer of value between users. These transactions are all public entries in the blockchain. The output of a transaction is an individual chunk of bitcoin currency that contributes to a user's balance if unspent. A user's balance is the sum of all of their unspent transaction output (UTXO).
Mempool	A local store of unvalidated transactions that can be packed into blocks by the miner subsystem.
Miner	Creates new blocks to be added to the blockchain by grouping transactions from the mempool into blocks. Miners must solve a difficult computation and the resulting "proof of work" is added to the block to prove its validity.
Blockchain	Provides Bitcoin's public ledger of transactions and is stored in a LevelDB database. It is a back-linked list of blocks of transactions that is the authority on what transactions have occurred.
Block	Another primitive subsystem that provides the basic structure of a block. Blocks are split into two parts: a header, and a list of transactions. The header contains a reference to the hash of the previous block in the blockchain, mining competition information, and the Merkle tree root.

High Level Reflexion Analysis

From our conceptual and concrete architecture diagrams, we can map the components as such:

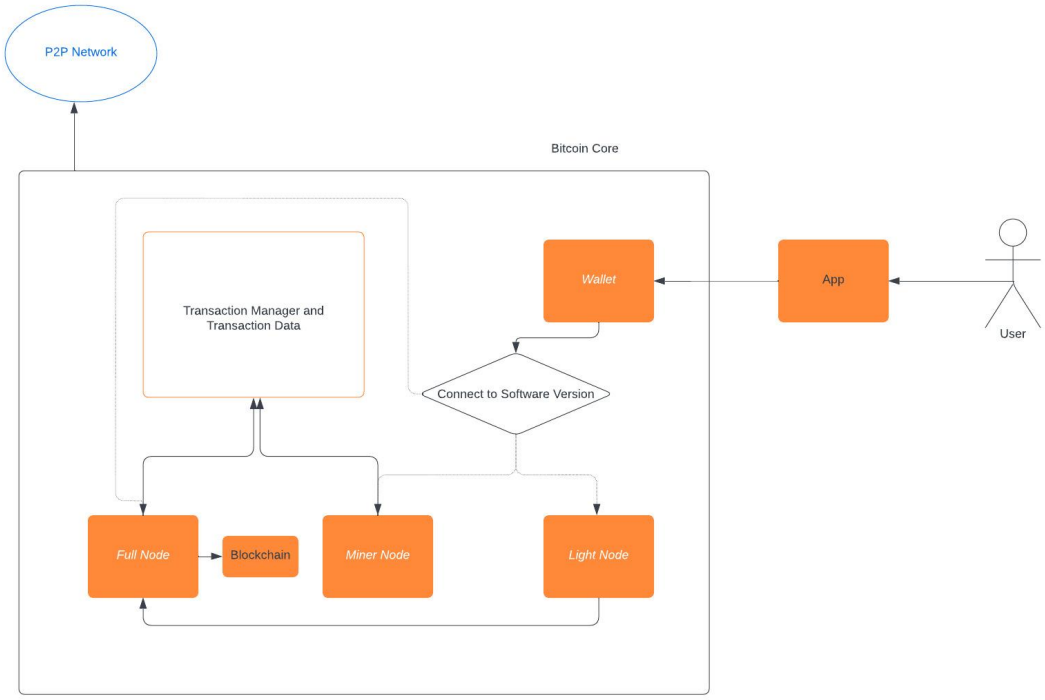

Figure 2. Conceptual architecture of Bitcoin Core from A1 for reference only

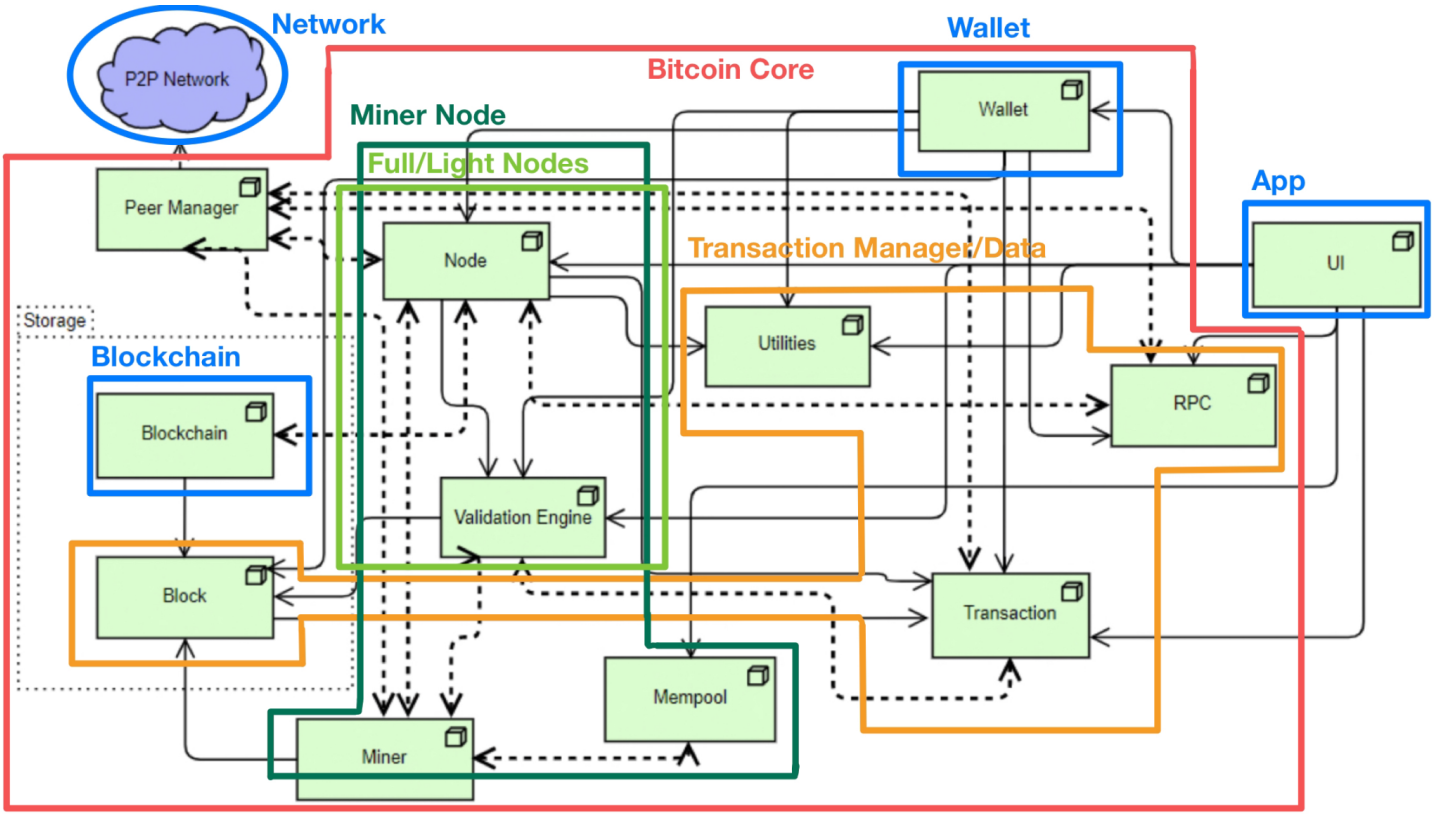

Figure 3. Conceptual architecture to concrete architecture mapping

There were far too many interdependencies to draw convergences and divergences, but our conceptual architecture includes all of the dependencies we found in the concrete architecture. Immediately, we could tell that our conceptual architecture was very simple compared to the concrete architecture, due to the increase in components. As such, it was difficult to map our conceptual architecture to the concrete because almost every component had dependencies on some other component. For example, the utilities functions were used by many distinct components, and the peer manager did not seem to fit in any component from the conceptual architecture.

Unexpected Dependencies

Due to the amount of unexpected dependencies, we chose the three most significant divergences to analyze. Most remaining dependencies were simply from components that had subsystems we could not predict.

UI Dependencies on Wallet, Node, Transaction, Mempool, and Utilities:

The sections of the code in the UI subsystem that depend on the other subsystems are largely sections and function calls that deal with initializing parts of the system once the user opens the GUI. Additionally, dependencies on the wallet and on the transaction subsystems are used so that the user's balance and keys can be viewed in the UI and transactions can be created and received.

Peer Manager and Miner Interdependency:

The peer manager and miner subsystems' interdependency was another one of the concrete architecture dependencies that we did not anticipate in our conceptual architecture. After investigating this dependency, we found that they rely on each other to update the Bitcoin network when new blocks are created to be added to the Blockchain. Additionally, the miner is responsible for the mathematical calculations that represent bitcoin's currency.

Wallet Dependency on Validation Engine:

The wallet subsystem depends on the validation engine subsystem as well. In our conceptual architecture, this was a relationship that occurred through other components instead of directly between the two. The wallet handles creation of transactions and keeps track of the user's balance. To do this, transaction parameters must be verified, thus the relationship with the validation engine. In our conceptual architecture, the wallet connected to the validation engine through the RPC and node subsystems. However, the source code contains a direct dependence from the wallet to the validation engine.

Top-Level Subsystem Analysis: Blockchain Storage

Diving further into the architecture of Bitcoin Core, our team chose to investigate the Blockchain Storage subsystem. This subsystem's main function is to keep a secure and decentralized record of every transaction that has occurred on the Bitcoin network. This is beneficial to Bitcoin Core because using a decentralized ledger means that it's maintained by a network of independent nodes rather than just one big central authority. Using the network makes it more resilient to censorship and tampering attacks.

Both our conceptual and concrete architectures came to the conclusion that the Blockchain Storage subsystem followed the Peer to Peer architectural style since it is a decentralized network of nodes that are both clients and servers. Each node stores a copy of the blockchain on its own computer and any new blocks to be stored are broadcasted to the entire network.

On initial startup or when a user's copy of the blockchain is far behind the tip of the best block chain, Bitcoin Core will perform a download of the entire blockchain from the network. This copy of the blockchain is stored in the data directory, where all of Bitcoin's data files are stored such as wallet data. In this directory there are two folders of interest in relation to blockchain storage: blocks and chainstate. The blocks folder contains; the actual Bitcoin blocks, a LevelDB database that contains metadata about all known blocks and how to find them, and a file for "undo" data. The chainstate is a LevelDB database of unspent transaction outputs and contains metadata about the origin of these transactions. This metadata is necessary for validating incoming blocks and transactions, and while validation is possible without it, it would require a full scan through every block for every output being spent.

Table 2. Subsystem Components and their responsibilities

Component	**Responsibilities**
RPC	The Remote Procedure Call (RPC) subsystem handles sending commands to remote servers and receiving any feedback. In the case of the storage subsystem, the RPC is used to receive data from other full nodes and send this data to be stored through the Storage Engine.
Storage Engine	The operating system's way of storing data. The exact specifications vary between systems, but in the case of Blockchain Store, it always involves the storage of blocks through either custom files or simple text documents.
Headers	Consists of three sets of metadata: A reference to the previous block hash, data related to mining (difficulty, timestamp, nonce), and the Merkle tree root which is a data structure used to summarize all the transactions in the block.
Blocks	A container data structure that aggregates transactions for inclusion in the blockchain. Composed of a header containing metadata, followed by a long list of transactions that make up the bulk of its size.
Validation Engine	Checks each block of transactions it receives to ensure that everything in that block is fully valid, allowing it to trust the block without trusting the miner who created it.
Utilities	This subsystem does not have one single responsibility or purpose but rather it is a collection of files/functions that provide common utility functions such as serialization, deserialization, and system settings.
Mempool	A "waiting area" for Bitcoin transactions that each full node maintains for itself. After a transaction is verified by a node, it waits inside the Mempool until it is picked up by a Bitcoin miner and inserted into a block.

Conceptual Architecture

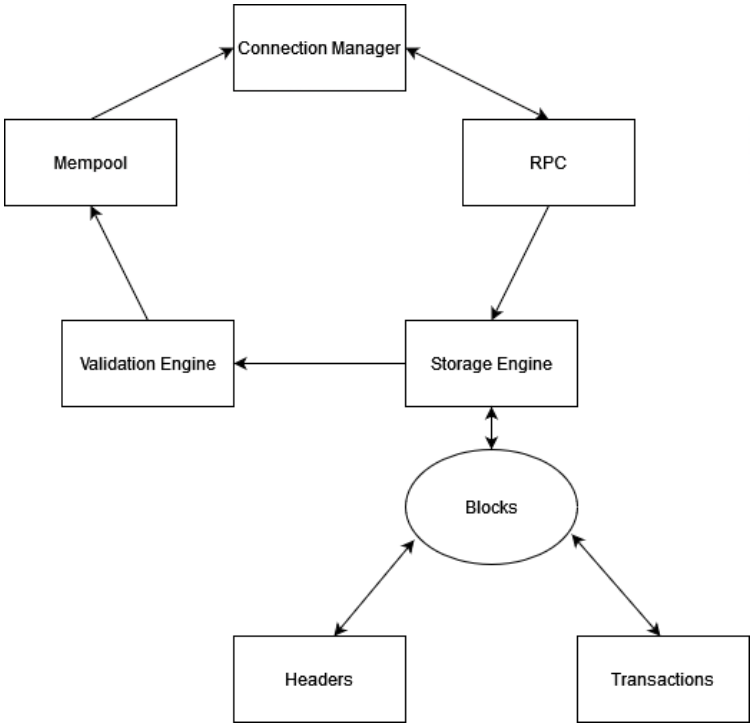

Figure 4. Conceptual architecture of Blockchain Storage

Based on this information, we derived the conceptual architecture that can be seen above. The conceptual architecture shows that when the user first loads the program or when their copy of the blockchain is far behind the tip of the best blockchain, which is the most-difficult-to-create chain, the program attempts to synchronize itself to the network by downloading new blocks. These blocks are then stored on the system's storage and contain a header with important metadata and a list of transactions. If the user wishes to validate these blocks and subsequently add them to the blockchain, they must be sent to the validation engine, which validates every block. After this is complete the blocks are sent to the Mempool and wait until they are picked up by a Bitcoin miner and added to the Blockchain.

Subsystem Reflexion Analysis

While analyzing the conceptual and concrete architecture of the Blockchain Storage system we found some differences in our concrete architecture compared to what we had done in our conceptual architecture in Assignment 1. In the conceptual architecture, we were able to identify all components of the blockchain storage system except for the mempool. The mempool is a key component since it is where verified transactions go before it is picked up by a miner and inserted into a block. Previous to this we only had a general idea of where the blockchain data went but we didn't have a proper name until we analyzed the concrete architecture in this assignment. In our reflexion analysis we also found unexpected dependencies, which are listed as the following.

Unexpected Dependencies

Utilities → RPC: Utilities as a subsystem does not serve a single responsibility or purpose, but rather as a collection of files/functions that provide common utility functions such as serialization, deserialization, and system settings.

Chainstate → Storage Engine: The Chainstate is a subsystem that we discovered in our investigation of the source code. It is a representation of all unspent transaction outputs and also contains metadata about previous transactions, which it uses to validate incoming blocks and transactions.

Storage Engine → Chainstate: The storage engine utilizes the CBlockIndex class that is created with the chainstate file. On a conceptual level, the storage engine needs the chainstate for its validation of blocks and transactions.

Validation Engine → Chainstate: The validation engine provides a multitude of methods that the chainstate uses to validate incoming blocks and transactions.

Validation Engine → Storage Engine: A multitude of methods used in blockstorage rely on a mutex declared in the validation file. A mutex is used to guard access to validation specific variables such as reading or changing the chainstage, or updating the transaction pool.

Mempool → Validation Engine: Transactions can be removed from the mempool for various reasons such as expiring, conflicts with in-block transactions, size limits, etc. These transactions are sent back to the validation engine to be re-evaluated and are either re-added to the mempool or deleted.

Transactions → Validation Engine: The validation engine utilizes the class CTransaction and its related methods which are declared in the transaction file. From a conceptual level this fits because the purpose of the validation engine is to validate the transactions that are found within blocks, and the logical way to do this on a concrete level is to use the same class and methods.

RPC → Mempool: The Mempool utilizes multiple functions and a struct found in RPCArg for different purposes such as submitting a raw transaction to a local node and network, conversion of transactions to JSON, or returning all transactions ids currently in the Mempool, among others.

Validation Engine → Blocks: The validation engine utilizes the block class and its various methods, namely CheckBlockHeader and TestBlockValidity. These methods check that the proof of work matches the claimed amount provided and checks the validity of a block through consensus respectively.

Mempool → Transactions: The mempool uses types that belong to the transaction file. This also follows the conceptual architecture given that transactions reside in the mempool until they are picked up by a miner and added to the network.

Concrete Architecture

With these dependencies uncovered, we were able to derive the conceptual architecture below.

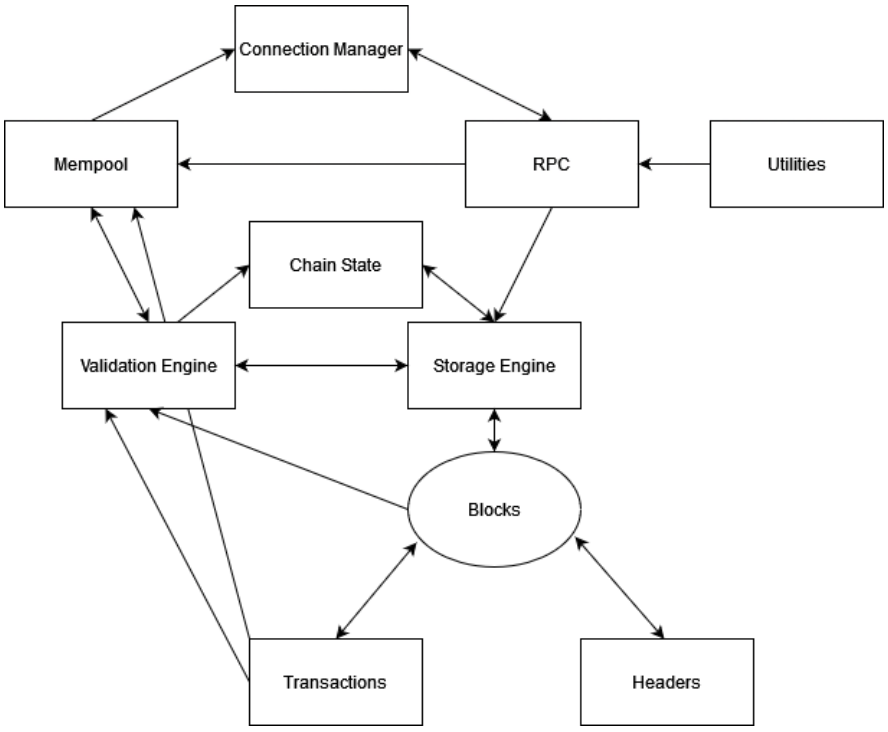

Figure 5. Concrete architecture of Blockchain Storage

External Interfaces

During the life of a Node (Bitcoin Core Client), it will have to send and receive numerous types of messages in order to fulfill its role in the Bitcoin network. Initially, when a node creates an outgoing connection, it advertises its version to the network using a `version` message. Until a remote node responds with its version, no other messages can be exchanged. If the remote node responds with a `verack` message, meaning Version Accepted, then interactions may continue.

Once this "handshake" is complete, there are numerous messages that can be sent and received by the block to carry out its desired role. These messages are outlined in Table 1.

Table 3. Bitcoin Client Messages

Message Type	Description
version	Message containing version number
verack	Message stating that version number is accepted
mempool	Request verified but unconfirmed transactions
getdata	Request content of specific object
inv	Message containing the "inventory" of node's known objects

notfound	Message stating object not in inventory
tx	Message containing info about a transaction
getblocks	Request blockchain
block	Message containing information about a block
getheader	Request a block's header
sendheaders	Request blockchain headers
headers	Message containing block headers
getaddr	Request information about objects on network
addr	Message containing information about nodes on the network
ping	Request to verify TCP/IP connection is still valid
pong	Message confirming TCP/IP connection is valid
reject	Response stating that the received message was rejected

These messages can be partitioned into two categories. These are "requests" and "responses". To illustrate this, Table 2 displays a list of Requests, each paired with all its possible responses. Note that all requests can have the possible response of `reject`. Furthermore, "[ ]" indicates that the response may be a list of the specified message.

Table 4. Requests and Responses

Request	**Response**
version	verack
Mempool	inv
getdata	inv, tx, headers, block, notfound
inv	getdata, getblocks, getheader, sendheaders
getblocks	block[]
getheader	headers
sendheaders	headers[]
getaddr	addr[]
ping	pong

Through this synthesis of the Bitcoin Core Client messages, we can clearly see how each node performs validation and proliferation of transactions, using these messages. For example, once a node has joined the

network by sending `version` and receiving `verack`, they may perform a transaction. After this they may receive a `getdata` message from a miner. The node would then respond with `inv`, and the miner would request the transaction with `getdata` again. From there the miner would verify the transaction – creating a new block – and then broadcast this info with an `inv` message. Finally, other nodes would send `getblocks` to retrieve the newly generated block from the miner, who would respond with a `block` message to send the block to the nodes and allow them to add it to their blockchain.

Use Cases and Diagrams

Use Case 1: Blockchain Storage (storing a new transaction)

One of the main reasons to use Bitcoin Core is safe transactions. Our first use case involves a new transaction between two users A and B being verified and stored. When a transaction is first received by a node, it is validated by the validation engine and added to the node's memory pool (the mempool). Here, the node updates the UTXO (Unspent Transaction Output) set to match the new transaction outputs that haven't been spent. Then, when a new block is mined the node will validate all transactions in the block. Once a valid hash is found, the miners create a new block containing the valid transactions. This block is broadcasted to the network and added to the blockchain by every node. The data of this transaction is stored in the block object as part of the block's Merkle tree structure. Once all of this is complete, the UTXO set is updated the second the transaction output has been spent.

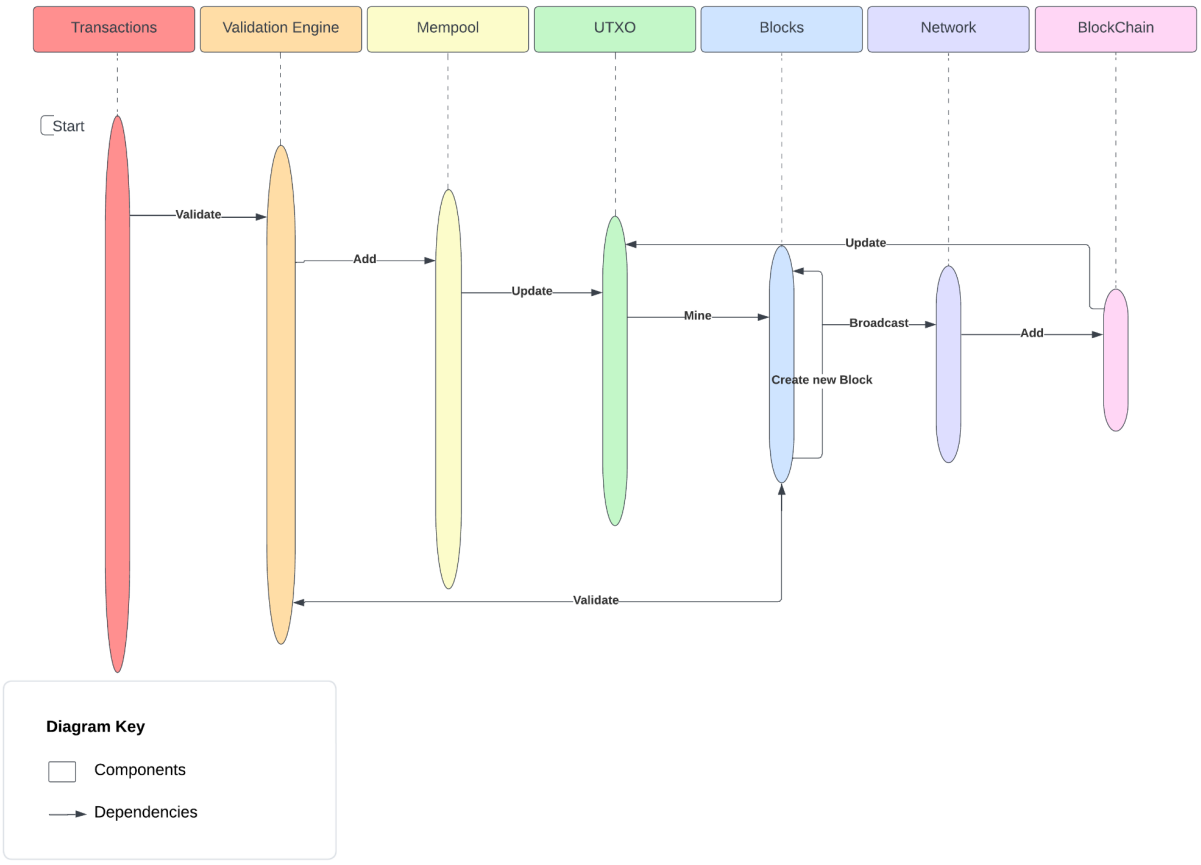

Figure 6. Blockchain storage use case sequence diagram

Use Case 2: Mining

In our previous report, we covered the mining use case, which is the process of earning and creating bitcoin through solving complex equations that verify transactions in a given block. We created an improved use case for mining using our new concrete architecture. The user sends a mining request that the RPC receives, and broadcasts to a node. This node then provides the network services to mine Bitcoin, which involves receiving transactions from the mempool and completing computations. The node will then update the blockchain using the peer manager, and this will be broadcast to the network. On success, the peer manager notifies the node, which will notify the user when the RPC connects the node to the UI.

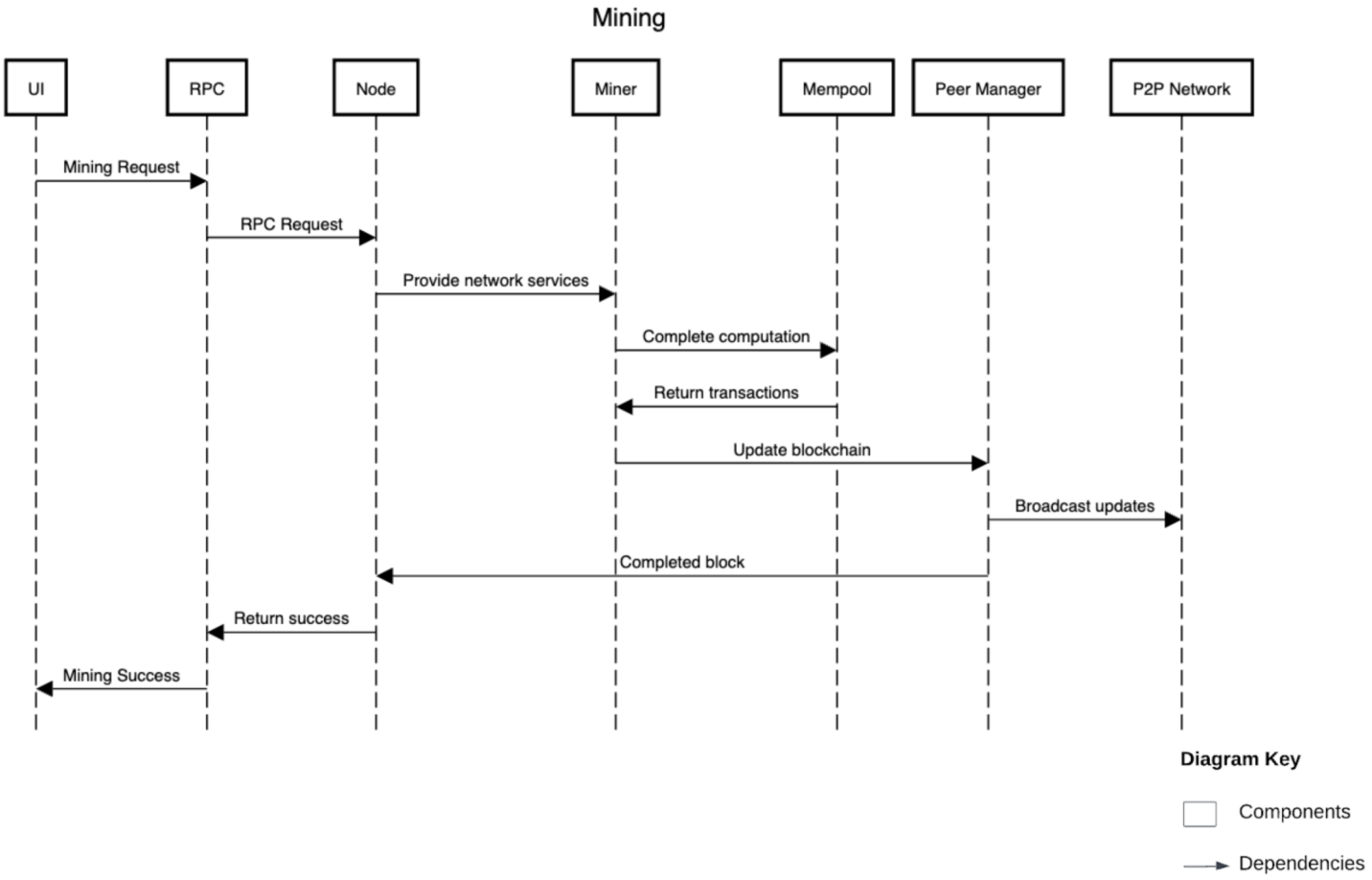

Figure 7. Mining use case sequence diagram

Data Dictionary

Block	A container data structure that schedules transactions for inclusion in the public ledger, the blockchain; a block is made up of two components: a header and a body
Fee	The amount remaining when the value of all outputs in a transaction are subtracted from all inputs in a transaction; the fee is paid to the miner who includes that transaction in a block.
Node	A computer running the Bitcoin software
Wallet	An application that serves as the primary user interface, controls access to the user's money, manages keys and addresses, tracking their balance, and creating and signing transactions

Naming Conventions

GUI	Graphical User Interface
UI	User Interface
P2P	Peer-to-Peer
RPC	Remote Procedure Call
UTXO	Unspent Transaction Output

Conclusion

Using Bitcoin Core's source code to visualize dependencies in Understand, as well as reading through the documentation, we were able to create a concrete architecture of Bitcoin Core. The team discovered that the core of the architecture is built around a peer-to-peer style but utilizes an object-oriented style in code. Further analyzing the systems through a reflexion analysis, we refined our understanding of the Bitcoin Core architecture and created two different use cases to better demonstrate the interaction of subsystems during the system's use.

Lessons Learned

Deriving the concrete architecture of Bitcoin Core involved a great deal of reading code and struggling to find documentation or comments that would help us to discern which component the file was a part of. When we began this process, we initially had difficulties with using Scitools' Understand software. Thus, if we were to redo this process, we would start off by exploring Understand and gaining a solid knowledge of what it can do would be an excellent first step. However, in the end, we learned to harness the capabilities of Understand, which proved invaluable in visualizing the connections between subsystems.

During our analysis, we discovered that many subcomponents of the Bitcoin Core concrete architecture actually exhibit an Object Oriented style, as opposed to the Peer to Peer style. We overlooked this in our conceptual architecture report. Although the entire system itself is definitely Peer to Peer, the actual inner workings are interdependent - characteristic of the Object Oriented style.

These divergences were a common theme with our conceptual and actual architectures. We found it almost unreasonable to map our conceptual architecture to our concrete architecture due to the vast amount of divergences and dependencies that we were not aware of. We learned that concrete architectures may rely on many interdependencies and subsystems that a conceptual architecture simply cannot predict until the development stage.

In terms of group work, the undertaking of deriving the concrete architecture was a big step up from the conceptual architecture, and we quickly realized it would take much more time than A1. Despite our busy schedules, we learned how our group works together from A1, so it was much easier to delegate tasks, ask for help from team members, and have everyone on the same page for A2.

References

https://cypherpunks-core.github.io/bitcoinbook/
https://github.com/bitcoin/bitcoin/tree/master/src
https://bitcoin.org/en/full-node
https://github.com/bitcoin/bitcoin/blob/master/doc/files.md
https://en.bitcoin.it/wiki/Data_directory
https://en.bitcoin.it/wiki/Protocol_documentation#Message_types