CISC 322/326 Assignment 2

CONCRETE ARCHITECTURE OF BITCOIN CORE

Group 8: EVANGELION-01

Jinlun Zhang: 16jz61@queensu.ca

Jing Tao: 21jt35@queensu.ca

Jihong Zhang: 19jz138@queensu.ca

Xueyi Jia: 18xj8@queensu.ca

Hejin Wang: 15hw45@queensu.ca

Yunhan Zhou: 17yz101@queensu.ca

MARCH 24th, 2023

1.0 Abstract

This report will recover the concrete architecture of Bitcoin Core by possibly adding and removing new and existing subsystems and dependencies on conceptual architecture in the previous report. The concrete architectural system of Bitcoin Core we determined is originated from mapping the source files to the conceptual components in the Scitools Understand, and we realized that its architecture style is still Client-and-Server for each node instance in the Peer-to-Peer Bitcoin Core network, which is the same style of conceptual architecture. This is because the basic operation of information transactions, requests and service responses transmission are not changed in the concrete architecture. However, there are a few missing dependencies in the conceptual architecture needed to be reflected, as we have found some valid gaps between the conceptual and concrete architectures after adding two more top-level components, “Utility” and “Consensus Method”, into the picture. Thus, a detailed reflection analysis is conducted. Besides that, we will choose the Miner to be our level-two subsystem for analysis. Finally, we will present 2 use cases and sequence diagrams to show the operations in the concrete architecture for the reflection model.

2.0 Introduction and Overview

Previously, we talked about the conceptual architecture of Bitcoin Core in A1 (Figure 1), now this report will deeply discuss the analysis of concrete architecture in Bitcoin Core system. The differences between conceptual architecture and concrete architecture are obvious, the concrete architecture attention to how conceptual architecture actually implements and executes in reality. Most of time, many dependencies and subsystems in conceptual architecture and concrete architecture are not matched, so we need to explain what functions or dependencies should be added or removed in concrete architecture to help complete tasks, this is the reason why we work a lot for reflection analyses. By examining code and testing on Understand, we added more models and unexpected dependencies. There are 14 essential components in total with the Client-and-Server architectural style to implement the huge service network that we confirmed, which will be comprehensively explained. Then we review our 2 use cases and sequence diagrams in the previous report to make some modifications based on the updated conceptual architecture. In the end, the operation of Bitcoin Core system will be concluded, and we will also show the limitations and what we learnt from working on the concrete architecture.

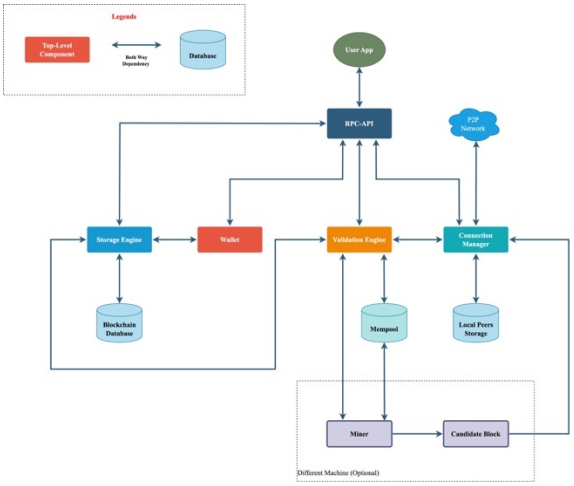

Figure 1: Original Bitcoin Core Conceptual Architecture (Client-Server Style)

3.0 Derivation Process

It is necessary to share how we successfully derived the concrete architecture, all in all, this should be inseparable from brainstorming, external knowledge research, organization and unifying ideas, and confirmation draft for the architecture.

We racked our brains to come up with Bitcoin Core conceptual structure, and then confirmed most of our conjectures through the open-source code of GitHub and the sketch we built on the Understand software by the source files provided on OnQ. Below is our uncleaned concrete architecture after we have mapped all files to the components in Understand (Figure 2), although our group members had our own controversies and opinions on the structure of diagram, we still selected the best one as the final architecture by communication and revision iteratively. After that, it was surprised to see the dependencies and arrow direction between components on the concrete architecture were not too different compared to the conceptual architecture in previous report. The cleaned concrete architecture we mainly need to rebuild is on the server side, so we add two components “consensus method” and “utility”, which are almost dependent with all the other components through uncleaned architecture in the Understand. To analyze how components operate, transform, and completes tasks for clients, we researched each module about what they should depend on and how they affect each other, especially for “consensus method” and “utility”. The process might be very cumbersome, and even some repetitive work was required. For example, we have to view W4 Approach (when, which, why, who) and discuss our findings together. This is an essential step in reflection model for the accurate rationale for dependencies between the subsystems.

As for selecting a 2nd-level subsystem, we analyzed components related to Miner, because it consists of collection of miners, hardware, and software, and they overwhelmingly host critical systems throughout Bitcoin Core for maintaining network. After finishing those, we found the rationale for discrepancies of conceptual and concrete architecture for reflection model and drew the final conclusion.

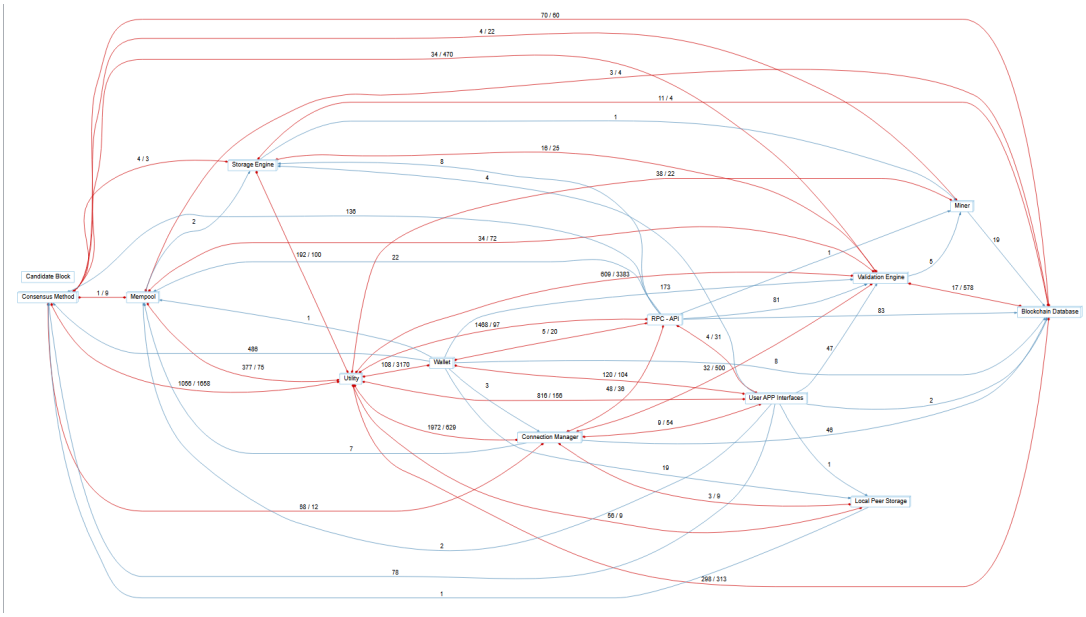

Figure 2: Uncleaned / Raw Concrete Architecture Obtained by Understand

4.0 Architecture Analysis

4.1 Concrete Architecture Components and Interactions

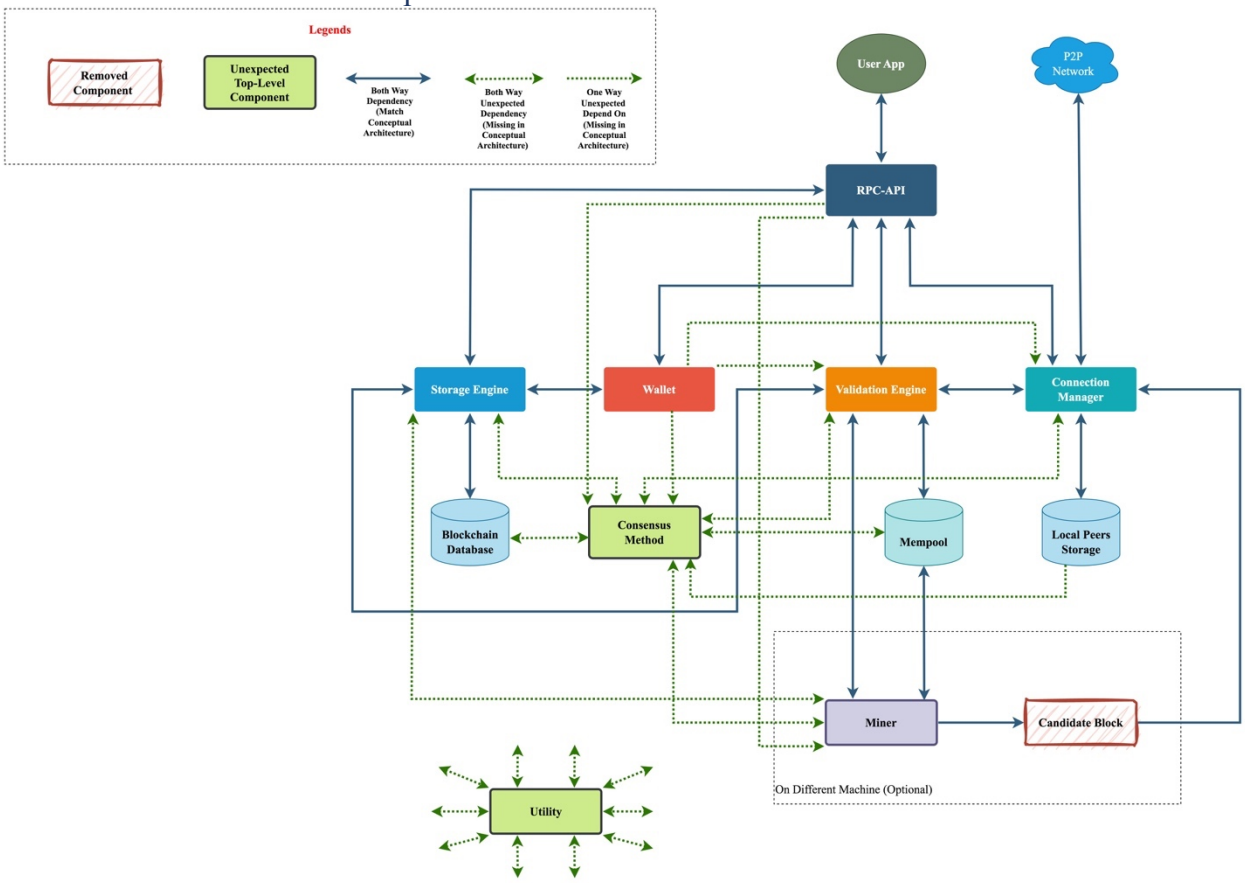

Figure 3: Bitcoin Core Cleaned Concrete Architecture

The original (raw / uncleaned) concrete architecture, as shown on Figure 2, is first acquired through mapping the source code files to the corresponding top-level conceptual components we proposed using Scitools Understand. However, since such uncleaned concrete architecture is complicated and contains a number of redundant dependencies (such as the dependencies created for the testing and maintenance purposes), we decide to refine the raw concrete architecture by hiding the 'invalid' dependencies (such as the dependencies that do not fulfill any functional requirement of Bitcoin Core) to improve the readability of our architecture diagram, and the outcome of the refinement is shown on the Figure 3.

Before moving into the detailed discussion about the concrete architecture of Bitcoin Core, we want to clarify certain assumptions we made about the cleaned concrete architecture. Firstly, we assume that the user is accessing and interacting with the Bitcoin Core node / system remotely through the RPC-APC component exclusively, namely that the User App Interface (GUI) is unneeded to be located on the same machine where the actual Bitcoin Core node resides. This explains why the User App Interface component only interacts with the RPC-API component,

while the RPC-API component has dependencies on all the remaining top-level components (except the database components), because the User App Interface component is expected to have direct dependencies on all the top-level components in the architecture (namely the administrator should have the authorization to access any component in such system), and so, as the direct agent of the user (administrator), RPC-API should accordingly demonstrate all the connections to all the components in the cleaned concrete architecture. Moreover, the assumption of remote access to the Bitcoin Core through User App Interface can effectively support the client-server software architectural style we proposed.

Secondly, to clean the concrete architecture for readability and to only preserve the meaningful dependencies, we assume that the unexpected dependencies for the run-time efficiency (usually one-way dependencies) are invalid and should not be reflected in the cleaned concrete architecture. This means that the top-level components like Wallet and Miner should not be able to bypass the database managers like the Storage Engine (managing the Blockchain Database) and the Connection Manager (managing the Local Peers Storage) to directly retrieve the information they need. As a result, the dependencies to either the database managers or the databases themselves can be aggregated into the dependencies to the database managers, simplifying the view of the concrete architecture.

Through observing the clean concrete architecture shown on Figure 3, one can immediately notice the 2 unexpected top-level components, ‘Utility’ and ‘Consensus Method’. The Utility component in the cleaned concrete architecture stores a wide variety of files for the consistent and reliable deployment and execution of Bitcoin Core. It not only centralizes a set of common and low-level functions that can be easily accessed and reused by different top-level components of the Bitcoin Core to complete tasks related time (ex: time.h), file I/O (readwritefile.h), and cryptography (ex: bytevectorhash.h), which avoids the potential massive code duplications and ensures the code consistency across the entire codebase, but also includes the functions relating to the compilation of Bitcoin Core and setting up the environment for the execution of Bitcoin Core. For example, there is a file called translation.h in the Utility component for setting up the user's native language in GUI. Moreover, the Utility component also includes the required functions for developer testing (ex: bench and test directories in scr), performance optimization, and system initialization (ex: init directory in scr). Thus, the Utility component serves as a comprehensive and versatile library for the overall operation of a Bitcoin Core node on the P2P network, which possesses 2-way dependencies on all the components in the architecture of Bitcoin Core.

As for the ‘Consensus Method’ component, it consists of files specifying the current rules of transaction validation and of the block addition to the Blockchain Database, leading all nodes on the P2P Bitcoin Core network to agree and stay on the same state of the public ledger (Blockchain) without conflicts. This explains why the Consensus Method component has code-level dependencies on both Storage Engine component and the Validation Engine component. In addition, though we have previously stated that the top-level components should not bypass the database managers to access the databases, we preserve the direct dependencies on the databases for the newly added Consensus Method component in the cleaned concrete architecture to explicitly illustrate how the new component interacts with the existing components, which also provides implications for the possible simplification for the following updated conceptual architecture of Bitcoin Core.

Through further investigations (with the help of git commit messages) on the dependency Consensus Method has on the Connection Manager, we validate the dependency between the 2 components from 2 granularity levels. In the conceptual level, because the Consensus Method needs to maintain an up-to-date view of the P2P network state, such as knowing the current longest chain and the statuses of other nodes, to ensure that all the nodes are in the agreement of the current state of the blockchain (Oreilly, n.d.), Consensus Method needs to directly communicate with the Connection Manager to exchange the information and data with other nodes to ensure "consensus" in the network. Moreover, in the code-level, the file 'chainparams.cpp' in Consensus Method, which is responsible for configuring various consensus-related parameters for the Blockchain, is directly dependent on the file 'chainparamsseeds.h' in Connection Manager, which contains a list of known and trusted nodes in the P2P network, to initialize the P2P communications for 'Consensus' in the network. Thus, the validity of the 2-way dependency between the Consensus Method and the Connection Manager is justified and should be correspondingly reflected in the updated conceptual architecture of Bitcoin Core.

Through this 2-level dependency verification strategy, we are capable of validating the remaining dependencies of other existing components such as Wallet, RPC-API, and Miner have on Consensus Method. For example, the one-way unexcepted dependency from RPC-API on Consensus Method is validated through inspecting the source code files mapped to the 2 components. The file 'httpserver.cpp' mapped to RPC-API is aimed to enable remote clients to interact with the Bitcoin Core node using the HTTP protocol, and it is called a file 'chainparamsbase.h' residing in the Consensus Method. Since 'chainparamsbase.h' contains specific information about the parameters of the blockchain (such as the current difficulty of the POW algorithm for the current block being mined), 'RPC-API' needs to retrieve the information from the 'Consensus Method' component and present them on the User APP when requested. Thus, this one-way unexpected dependency should be considered as legitimate as it is not added for testing or performance reasons. Moreover, through exploiting the 'StickyNotes' approach and the 'Blame' git version control functionality on the source code, the instruction '#include <chainparams.h>' is found to be initially added 8 years ago for supporting the HTTP dispatch mechanism (according to the commit message) without any significant modification afterwards, which further enhances the credibility of our proposition that the one-way unexpected dependency is valid and should be correspondingly reflected in the updated conceptual architecture of Bitcoin Core.

Furthermore, it is noteworthy that the reason why the Consensus Method has a valid 2-way dependency on the Miner component is that the Consensus Method also includes the Proof-of-Work (PoW) algorithm needed to be eligible to receive the rewards of mining a new block, as well as the difficulty adjustment algorithm needed to maintain a relatively constant rate of new blocks being added to the blockchain, leading the dependency between the 2 components to be essential for the mining use case and the overall reliability of Bitcoin Core.

As for those unexpected dependencies between the existing components displayed in Figure 3, such as the one-way dependency from RPC-API to Miner, they are all manually verified (with the help of Scitools Understand and the approaches aforementioned) to be necessary for the reliable, consistent, and secure implementation and execution of the Bitcoin Core software on the user machine (as a node in the P2P Bitcoin Core network), which will be discussed in details in the Reflection Analysis.

Finally, it is notable that there is no missing dependency according to our cleaned concrete architecture, meaning that all the conceptual dependencies we proposed before are legitimate and can be supported by the file-level dependencies between the conceptual components. However, the Candidate Block component may be considered as missing because there is no source file that can be reasonably mapped to this component, implying the non-existence of Candidate Block component. Through reflection on our original conceptual architecture, we realize that this component was originally added for the illustration purpose of the mining use case, and so should not serve as a top-level component like Validation Engine, but a subsystem within the Miner itself. Thus, this component should be considered as invalid and be hidden in the updated conceptual architecture accordingly.

4.2 Updated Conceptual Architecture

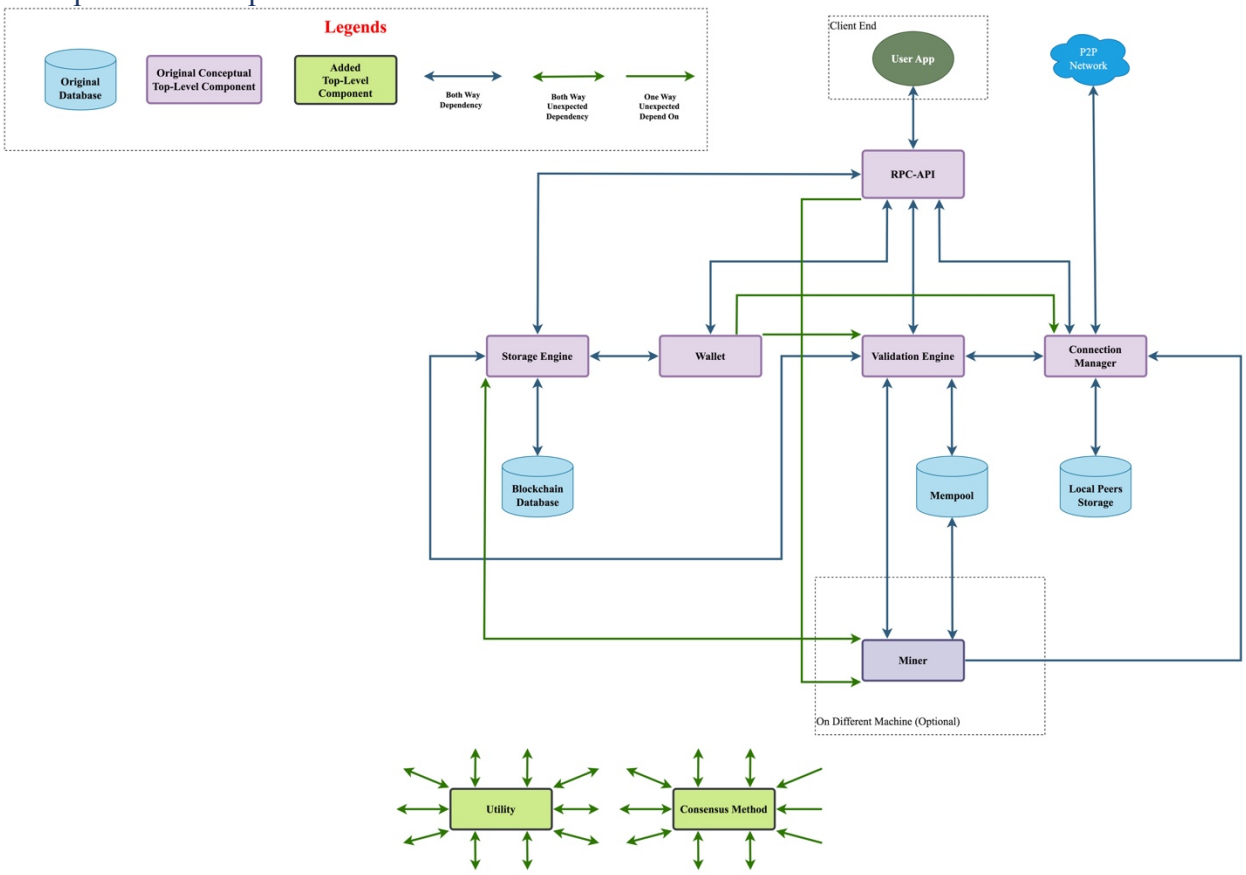

Figure 4: Bitcoin Core Updated Conceptual Architecture

The updated conceptual architecture is constructed based on the concrete architecture and further inspections of the source codes that formulate the unexpected dependencies in the cleaned concrete architecture. Though the ultimate goal should be re-designing our conceptual architecture to ensure an exact match with the cleaned concrete architecture, we decide to further simplify the cleaned concrete architecture for the readability reason and preserving only the essential top-level components and the dependencies that are required to fulfill the possible functional requirements of Bitcoin Core, such as the user should be able to observe certain key

parameters of the Miner on the User App Interface when requested (the one-way dependency from RPC-API on Miner). The result of this further simplification is shown on Figure 4.

As mentioned before, the Candidate Block component is not included in the diagram and the Consensus Method is separated from the main body of the architecture to be regarded as a utility-like component as it is nearly depended by all the remaining components in the architecture. As a result, the updated conceptual architecture is highly suitable for our client-server architectural style proposition, where the User (as the client) only interacts with the Bitcoin Core through the Internet and the RPC-API to send the user requests and commands to the rest of components / servers. In the client-server style, each component is considered to be a server that provides certain functionalities and do not have to reside on the same machine, and the database component must be combined with its corresponding data manager to represent a data server. Those style invariants are fitly captured by our updated conceptual architecture, where the Miner is usually located on a separate and dedicated mining machine (such as the application-specific integrated circuits (ASIC) that prints hundreds of mining algorithms in hardware, running in parallel on a single silicon chip (GitBook, n.d.)), and the RPC-API serves as the central register of server names and services provided to enable low response time and efficient collaborations between servers. Meanwhile, the 2 newly added components act as the general base servers that can be effectively accessed, requested, and utilized by the other components in the software.

However, since components interact with each other through method calls (include) and the data representation of one component / object is hidden from the other objects (only be accessed or modified through the official API of each object), one may argue that the architectural style can also be considered as Object-Oriented. However, one should also notice that the objects / components in Bitcoin Core are unable to work completely independently and asynchronously. For example, the RPC-API is considered to be the central server and has to work with the other objects in the system such as the Wallet and the Consensus Method. Moreover, it frequently has to wait for the returns of other components. For instance, when the user wants to turn off the Miner, the RPC-API has to wait for the successful shutdown notification from the Miner. Moreover, some components such as Blockchain Database may not have established interfaces for communications between components, leading this style to be unsuitable for Bitcoin Core.

Therefore, through updating our original conceptual architecture of Bitcoin Core based on the cleaned concrete architecture and careful analysis, we are more determined to state that the architectural style of Bitcoin Core is likely to be Client-Server.

4.3 Reflection Analysis

As mentioned in Concrete Architecture Components and Interactions (4.1), there are no missing dependencies after comparing Updated Conceptual Architecture to our Cleaned Concrete Architecture. Therefore, the main goal of this reflection analysis is to analyze the four new unexpected dependencies according to our research.

RPC-API => Miner

The "Miner" component within Bitcoin Core is in charge of generating candidate blocks and incorporating them into the blockchain. In the code level, the dependency arrow shows that the “mining.cpp” includes <pow.h> and calls CheckProofOfWork in its file. In the conceptual level,

the RPC-API component has a dependency on the Miner component because it provides the interface for users and developers to access and control mining-related functionality. Through the RPC-API, users and developers can issue commands related to mining, such as starting or stopping the mining process, adjusting mining parameters, and obtaining mining-related information, such as the current mining difficulty or the status of the mining operation. This dependency ensures seamless communication and interaction between the RPC-API and the Miner component, enabling the user to manage mining operations effectively and remotely.

Wallet => Validation Engine

The Validation Engine component ensures transaction and block authenticity. It verifies transaction digital signatures, proper formatting, and adherence to Bitcoin protocol rules. Additionally, it confirms accurate block construction and appropriate linkage to the preceding block within the blockchain. In the code level, the dependency arrow shows that there are multiple files in Wallet component depending on Validation Engine, including “coincontrol.h” which includes <outputtype.h>, and “wallet.cpp” which includes <feerate.h> under the Policy subsystem. In the conceptual level, the Wallet component in Bitcoin Core has a dependency on the Validation Engine because it relies on the validation process to ensure the integrity and security of its transactions. Moreover, this dependency is crucial in maintaining a secure and accurate record of the user's funds and transactions. The wallet must be able to recognize and interact with valid transactions on the blockchain. By relying on the Validation Engine, the wallet can confirm incoming and outgoing transactions, avoid double-spending, and maintain an up-to-date balance.

Wallet => Connection Manager

Connection Manager component facilitates decentralized, peer-to-peer communication among Bitcoin Core P2P network nodes. It encompasses protocols for creating connections, message exchange, and handling peer interactions. In the code level, the dependency arrow shows that the “init.cpp” includes <net.h>, as well as the “walletdb.cpp” includes <protocol.h> in its file. In the conceptual level, the Wallet component has a dependency on the Connection Manager component because it needs to communicate with other nodes in the P2P network directly to send and receive transaction data. Moreover, the Wallet component relies on up-to-date information about transactions and the state of the blockchain to function correctly. Through the Connection Manager, Wallet is able to broadcast new transactions, receive updates about incoming transactions, and stay in sync with the network.

Storage Engine <=> Miner

The storage engine component manages blockchain storage and other data associated with the Bitcoin network. In the code level, similar to RPC-API, the dependency arrow shows that the “pow.cpp” includes <chain.h> and calls GetAncestor in its file, as well as returns like nBits and uses like nHeight. In the conceptual level, the Miner component depends on the Storage Engine component to access essential blockchain data for effective mining operations. The Storage Engine maintains the blockchain and related information, such as pending / orphan blocks. The Miner requires this data to properly link new blocks to predecessors, include unconfirmed transactions for validation and processing, and ensure adherence to Bitcoin protocol rules, like meeting the current difficulty target. This reliance on the Storage Engine enables the Miner to create valid blocks and bolster the network's security and consensus. In addition, since the

dependency is bidirectional, the Storage Engine component has dependency on the Miner component. In the code level, the “txdb.cpp” includes <pow.h> and calls CheckProofOfWork in its file. In the conceptual level, the Storage Engine needs to be timely informed of the new blocks generated by the Miner and cache them correctly so that the blocks can be effectively appended to the Blockchain Database once they have been validated throughout the P2P Bitcoin Core network.

5.0 Chosen 2nd-level Subsystem

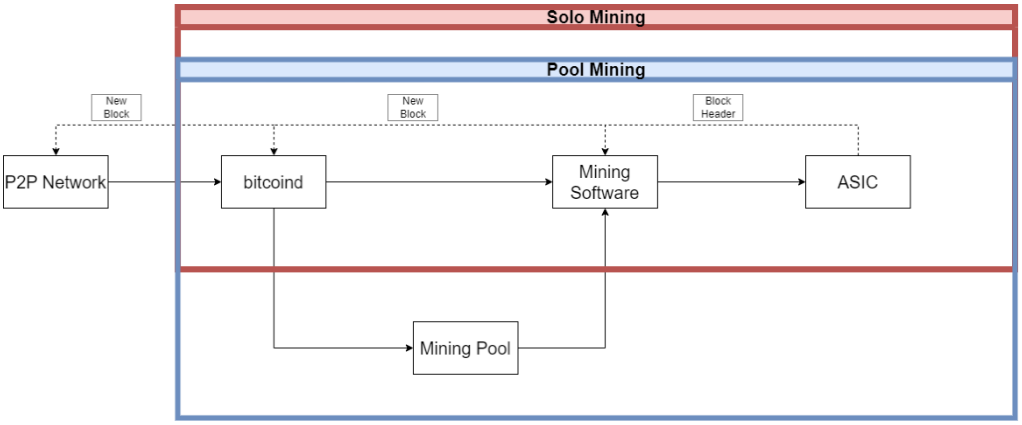

Figure 5. *Conceptual Architecture of Mining*

The architecture of mining of blocks and its interactions from a conceptual and concrete point of view will be discussed in this part. The mining subsystem in the Bitcoin network does not have a rigid or centralized architecture. Instead, it consists of a decentralized collection of miners, hardware, and software working together to maintain the network. However, we can outline a high-level, conceptual architecture of the mining subsystem (Figure 5):

Mining hardware: ASIC miners (Application-Specific Integrated Circuits): Most used for Bitcoin mining, offering the best performance and efficiency.

Mining software: This software interfaces with the hardware and connects miners to the Bitcoin network. It provides performance monitoring, optimization, and configuration capabilities. Examples include CGMiner, BFGMiner, and NiceHash.

Mining pools: Miners often join mining pools to combine their hashing power and share rewards proportionally. Pools improve reward consistency by distributing shares based on each miner's contribution. Examples of mining pools include Slush Pool, F2Pool, and Antpool.

Mining nodes: These nodes are responsible for validating transactions and blocks. They enforce the consensus rules, relay new blocks and transactions, and maintain a copy of the blockchain.

Networking and communication: The mining subsystem relies on the peer-to-peer (P2P) Bitcoin protocol to communicate with other nodes and miners. This protocol facilitates the broadcasting of new transactions, blocks, and other essential information.

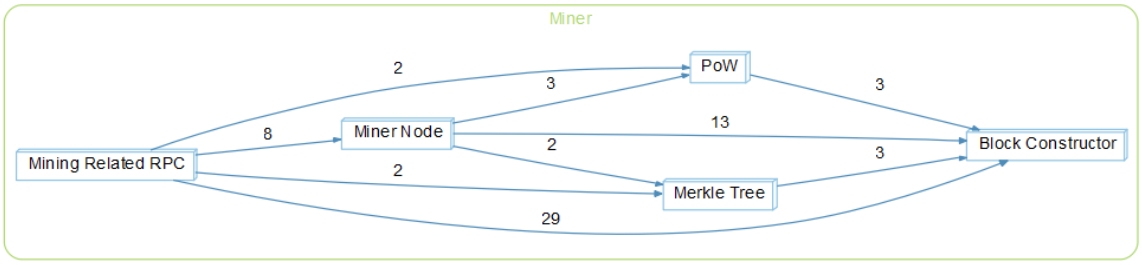

***Figure 6**. Dependencies of Mining by Understand*

From the dependencies of mining of block obtained by Understand, there are some new essential components shown in figure 6:

Block Constructor: This component is responsible for creating new blocks by gathering transaction data from the network, constructing the coinbase transaction, and assembling the block header. It communicates with the Miner Node and the Merkle Root components.

Proof of Work (Pow) Algorithm: This component implements the hashing algorithm (SHA-256) to find the nonce value that meets the target difficulty threshold. It receives the block header from the Block Constructor and iterates through possible nonce values until it finds a hash that meets the requirements. Then it returns the successful nonce to the Miner Node.

Miner Node: This component represents the mining software running on the miner's computer. It interacts with the Bitcoin network to obtain new transactions, communicates with the Block Constructor to create new blocks, and uses the Pow Algorithm to find a valid nonce. If a valid nonce is found, the Miner Node submits the completed block to the network.

Mining-related RPCs: This component handles communication between the Miner Node and the Bitcoin network (bitcoind) or mining pools. It includes the deprecated getwork RPC, the getblocktemplate RPC, and the Stratum mining protocol. These RPCs provide the Miner Node with the necessary data to create blocks and submit shares (in the case of pooled mining).

Merkle Tree: This component is responsible for creating the Merkle tree from the transaction data, adding extra nonce data to the coinbase transaction, and computing the Merkle root. The Merkle root is then used in the block header by the Block Constructor.

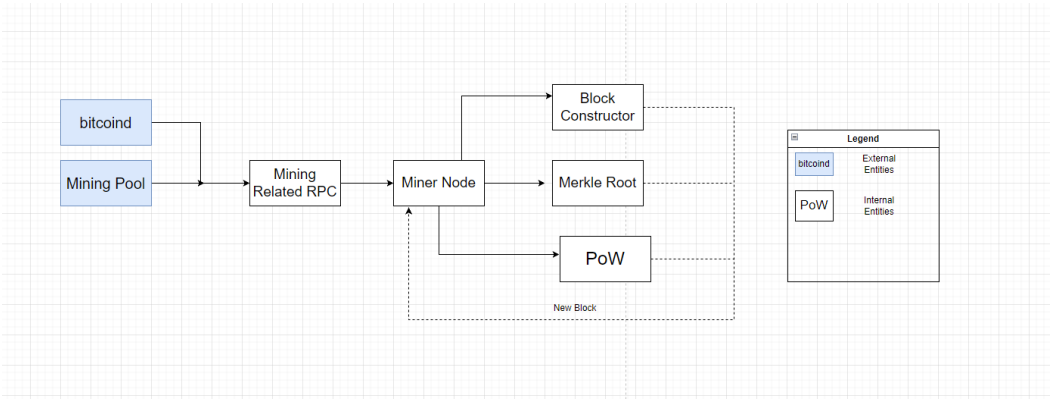

***Figure 7**. Concrete Architecture of Mining*

In addition to these components, the architecture (Figure 7) also involves two types of mining approaches:

Solo Mining: In this approach, the miner operates independently to generate new blocks. The entire block reward and transaction fees go to the miner, resulting in larger payments with higher variance.

Pooled Mining: In this approach, miners collaborate and pool their resources to find blocks more frequently. The rewards are shared among pool participants according to the amount of hashing power they contribute, resulting in smaller payments with lower variance.

The interactions and dependencies among these components are as follows:

- The Miner Node communicates with the Bitcoin network (bitcoind) or mining pools using Mining-related RPCs to obtain transaction data and submit completed blocks or shares.
- The Block Constructor uses transaction data provided by the Miner Node to create new blocks, incorporating the Merkle root generated by the Merkle Root component.
- The Merkle Root component computes the Merkle root based on the transaction data and updates it wheSn the extra nonce field in the coinbase transaction needs to be changed.
- The Pow Algorithm works with the Block Constructor and the Miner Node to find a valid nonce that meets the target difficulty threshold.
- The Miner Node submits the completed block (with the valid nonce) to the Bitcoin network (bitcoind) or mining pool if the Pow Algorithm finds a successful nonce.

Concurrency in the mining subsystem is an essential aspect of the Bitcoin network's functioning, as it allows multiple miners to work on finding new blocks simultaneously. Concurrency plays a vital role in maintaining the decentralized nature of the network, as it enables miners from around the world to compete fairly for block rewards. Here are some key aspects of concurrency in the mining subsystem:

- **Parallel mining:** Miners across the globe work independently and simultaneously on creating new blocks. They each select transactions from the mempool, create a block template, and try to solve the Proof-of-Work (PoW) puzzle. The parallel nature of mining ensures that no single miner or group of miners can monopolize the process.
- **Orphaned blocks**: Occasionally, two or more miners may solve the PoW puzzle and broadcast their blocks almost simultaneously. In such cases, the network might temporarily have multiple competing versions of the blockchain. Nodes will adopt the version with the most accumulated PoW (the longest chain) as the valid blockchain, causing the other blocks to become orphaned. Concurrency in mining can lead to such conflicts, which are resolved by the Bitcoin protocol's consensus mechanism.

6.0 Use Cases

Use Case 1:

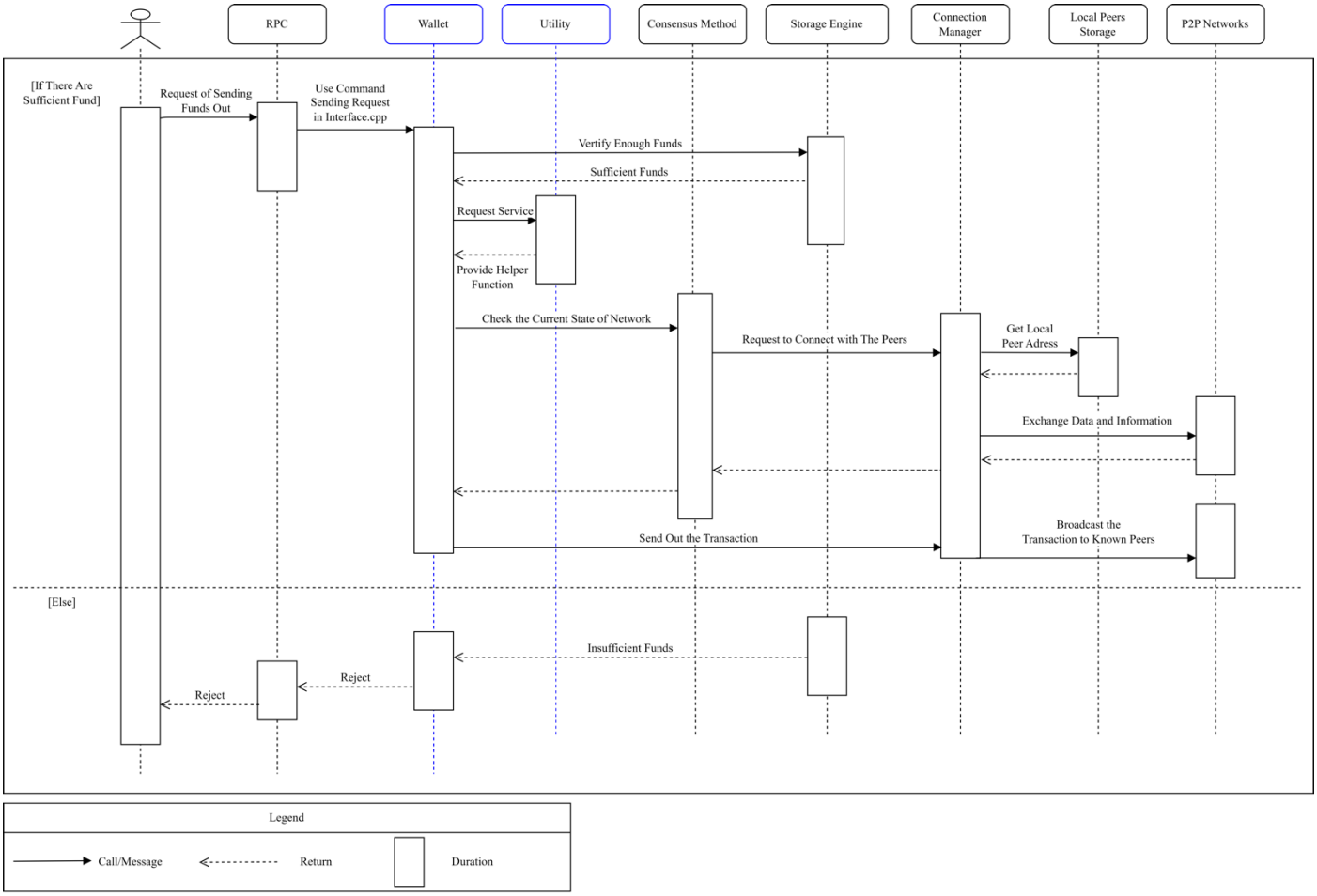

Figure 8: Sequence Diagram of Sending Funds Out

This use case is the process of sending funds. Firstly, the user initiates the transaction (sending funds out) by interacting with the RPC-API. Then the Wallet component receives the request from the RPC-API then it send the request to verify the user's credentials and balances in Storage Engine, ensuring that the user's account has enough funds. if it sufficient, then proceed, otherwise reject. If it has sufficient funds then return to Wallet, and Wallet send the request for Utility component for the functions facilitating the transaction. Then the Consensus is responsible for validating the transaction from the Wallet component, at the same time the Consensus is exchanging the information and data with the Connection Manager. Then the Wallet accesses the Connection Manager to send out the transaction. The Connection Manager will broadcast the transaction to the local peers that are stored in the local list of peers.

Use case2:

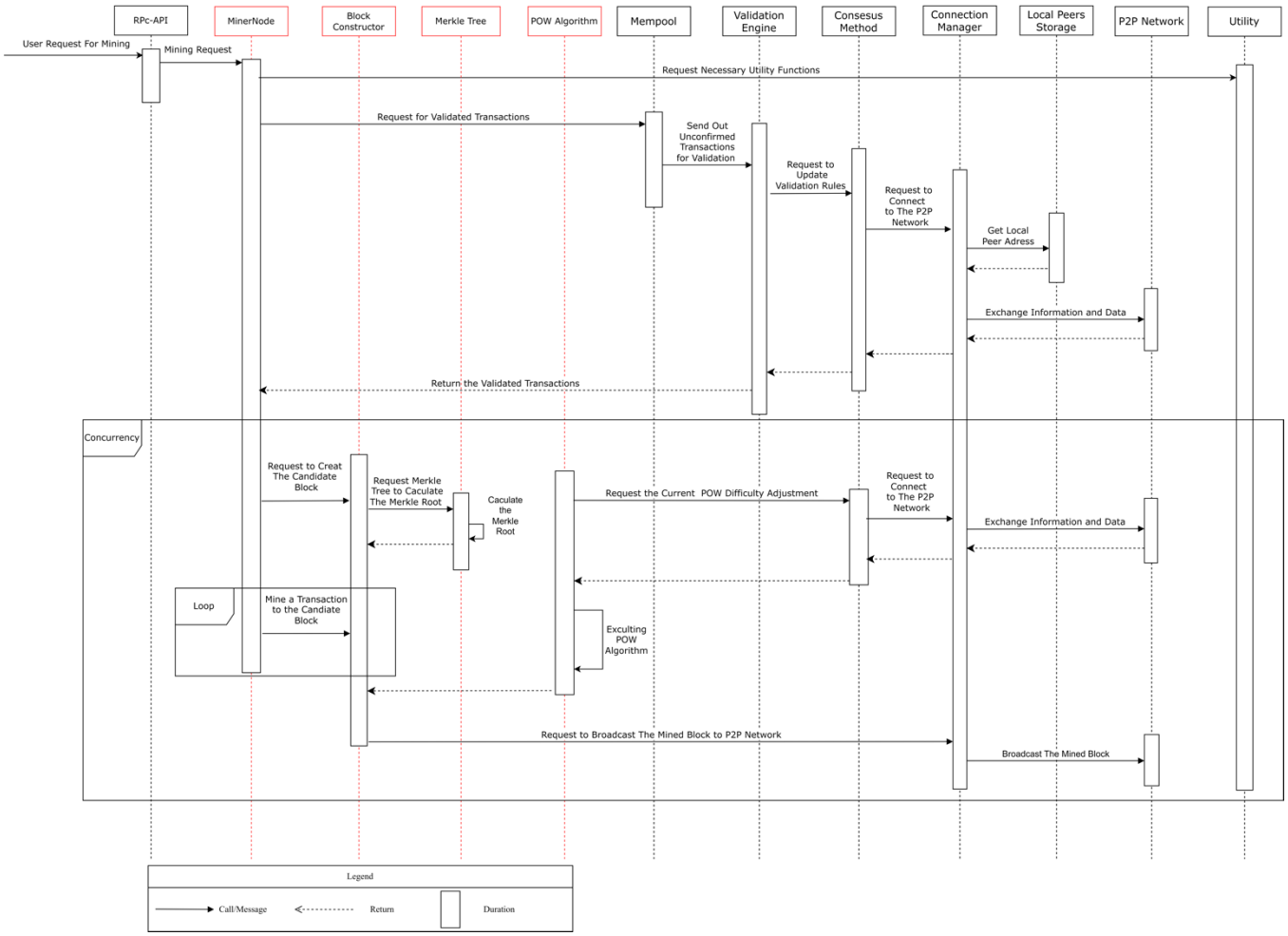

Figure 9: Sequence Diagram of Mining

This Sequence Diagram explains the process of mining. First, the user sends a mining request to the MinerNode through the RPC-API. MinerNode sends the request to the Utility component (and it will provide service for all components all time) for necessary functions and MinerNode communicates with the Mempool to request validated transactions. The Mempool sends out the unconfirmed transactions for validation to the Validation Engine, and the Validation Engine requests Connection Manager to connect to the P2P Network. Then the Connection Manager gets the local peers at the Local Peers Storaged exchanges information and data with the P2P Network. Then the validated transactions are returned to the MinerNode. Later, MinerNode communicates with the Block Constructor to create a candidate block, then Block Constructor requests the Merkle Tree component to calculate the Merkle root, Merkle Tree component calculates the Merkle root and returns it to the Block Constructor. At the same time, the POW (Proof of Work) Algorithm Request the Current POW Difficulty Adjustment from the P2P Network through the Connection Manager. Then the POW Algorithm executes the POW Algorithm and returns the result to the Block Constructor. After the MinerNode continuously appends the transaction to the candidate block, the Block Constructor completes the candidate block and broadcasts to the P2P network through the running Connection Manager.

7.0 Lessons Learned, Limitation and Difficulties Encountered

In delving into the complex software design of Bitcoin Core, we gained a great deal of insight. Despite adding the "Utility" and "Consensus Method" components, we still found some differences between the Conceptual Architecture and the Concrete Architecture. Many dependencies and subsystems do not match between Conceptual Architecture and Concrete Architecture. Some connections do not exist in the abstract design, and the implemented software exhibits additional functionality and uses different mechanisms than the conceptual plan.

An important takeaway was our experience using Scitools Understand to examine the interdependencies between components. This analysis allowed us to discern exactly what the purpose of each module was in the abstract architecture. Scitools Understand also exposed the surprisingly large size of the current code base. By using Scitools Understand, developing a concrete architecture based on a conceptual design is easier because self-analysis of the dependencies between modules is very difficult due to the complexity of existing software.

In addition, we deepened our appreciation of the value of teamwork in the process. When team members encountered difficulties using Scitools Understand, the group offered help. For example, we modularized the source code, individually analyzed the modules to which the source code belonged and worked together to establish dependencies on each other's components. In this work, the collaboration proved to have a tremendous impact.

In this case, the project had more limitations than our previous work. First, Scitools Understand is a complex tool, and using it to evaluate the interdependencies between modules within the Bitcoin Core software is often time-consuming. In addition, there needs to be more development documentation in the source code, and no ReadMe file is provided, which significantly increased our time to evaluate and discuss Bitcoin thoroughly and hindered our research into the precise architecture of Bitcoin Core.

Secondly, while investigating the Bitcoin implementation architecture, we found that analyzing the source code files into the corresponding top-level conceptual components via Scitools Understand complicated the Concrete Architecture because it contained redundant dependencies (e.g., some dependencies were created for testing and maintenance purposes). For this reason, we hide some non-functional dependencies in the final result to improve the readability of Concrete Architecture.

8.0 Data Dictionary

User App Interface: The graphical representation or front-end application through which users interact with a software system.

SciTools Understand: A software analysis tool used to analyze codebases and identify dependencies, architecture, and other essential aspects.

Subsystem: A smaller, independent component within a larger system, responsible for specific tasks or functions.

Dependency: A relationship between software components or modules where one relies on the functionality provided by another.

Blockchain Database: A decentralized digital ledger that stores transactions in chronological order, grouped into blocks, and maintained by a network of computers.

Storage Engine: A software component responsible for managing the storage, retrieval, and updating of data in a database system.

Connection Manager: A software component responsible for managing connections to other systems or components, such as local peers in a network.

NiceHash: A cryptocurrency mining marketplace that connects miners and buyers of hashing power.

F2Pool: A large mining pool that allows miners to combine their computational resources to mine cryptocurrencies more efficiently.

9.0 Naming Conventions

RPC (Remote Procedure Call) : a communication protocol that enables one program to request services from another program located on a remote computer.

APC (Asynchronous Procedure Call): a communication protocol that allows a program to request services from another program without waiting for the service to complete before continuing.

PoW (Proof-of-Work algorithm): A consensus mechanism in blockchain networks that requires participants to perform complex computational tasks to validate transactions and create new blocks.

ASIC miners (Application-Specific Integrated Circuits Miners): specialized hardware designed for the specific purpose of mining cryptocurrencies using the Proof-of-Work algorithm.

CGMiner (Conformal GMiner): A popular open-source, cross-platform mining software used for Bitcoin and other cryptocurrencies.

BFGMiner(Big Freakin' GMiner): The open-source mining software which similar to CGMiner, used for Bitcoin and other cryptocurrencies.

11.0 Conclusion

In this report, we focus on the direct dependencies between the specific architecture of Bitcoin Core and its subsystems. The client-server architectural style suits the Concrete Architecture of Bitcoin Core. In analyzing the Conceptual Architecture to derive a Concrete Architecture, we found that the initial diagrams in our previous reports needed to represent the actual conceptual architecture accurately. By studying the developer's documentation and using Scitools Understand for graphical visualization, we could develop a refined conceptual architecture and then use it to derive the Concrete Architecture. Reflective analysis revealed new dependencies between the various subsystems. And finally, with the previous conceptual architecture, we added "Utility " and "Consensus Method" components. We also built a concrete architecture to remove some dependencies for testing and maintenance to clarify the dependencies between

components. In addition, we specifically studied the mining subsystem in Bitcoin Core and built a high-level conceptual architecture for it.

Overall, we gained a deeper understanding of the system and its underlying architecture through careful observation and analysis of the Bitcoin Core architecture with the help of Scitools Understand. This more profound understanding of Bitcoin Core's specific architecture will prove invaluable for future software research, development, and maintenance efforts.

12.0 Reference

Binance Academy. (2022, December 12). *What is a blockchain consensus algorithm?* Binance Academy. Retrieved March 23, 2023, from https://academy.binance.com/en/articles/what-is-a-blockchain-consensus-algorithm

Antonopoulos, A. M. (n.d.). *Mastering bitcoin*. O'Reilly Online Learning. Retrieved March 23, 2023, from https://www.oreilly.com/library/view/mastering-bitcoin/9781491902639/ch08.html

Chapter 2: 'how bitcoin works'. Chapter 2: 'How Bitcoin Works' · GitBook. (n.d.). Retrieved March 23, 2023, from https://cypherpunks-core.github.io/bitcoinbook/ch02.html

Bitcoin: A peer-to-peer electronic cash system. Bitcoin. (n.d.). Retrieved March 24, 2023, from https://bitcoin.org/en/bitcoin-paper

Bitcoin.org. (n.d.). Bitcoin Core Developer Documentation. Bitcoin Core. (n.d.). Retrieved March 24, 2023, from https://bitcoincore.org/en/doc/

Bitcoin. (n.d.). *Bitcoin/Bitcoin: Bitcoin Core Integration/Staging tree*. GitHub. Retrieved March 24, 2023, from https://github.com/bitcoin/bitcoin

Luxembourg, A. B. U. of, Biryukov, A., Luxembourg, U. of, Luxembourg, D. K. U. of, Khovratovich, D., Luxembourg, I. P. U. of, Pustogarov, I., University, A. S., University, G.-- C., University, P., & Metrics, O. M. V. A. (2014, November 1). *Deanonymisation of clients in Bitcoin P2P network: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM Conferences. Retrieved March 24, 2023, from https://dl.acm.org/doi/10.1145/2660267.2660379