Bitcoin Core Concrete Architecture Report

CISC322

Thursday, March 23rd, 2023

Eric Lam – 19ewhl1@queensu.ca – 20229013

Andrew Zhang– 19az32@queensu.ca – 20210066

Dylan Chipun – 19dmc10@queensu.ca – 20224970

Amy Hong – 19yh94@queensu.ca – 20219853

Sueyeon Han – 19sh77@queensu.ca – 20217002

Asher Song – 17acds@queensu.ca – 20112257

ABSTRACT (AMY)

This report explores the detailed architecture of the Bitcoin coin. This includes deriving the concrete architecture from Bitcoin Core's source code, comparing and contrasting this new architecture with a revised conceptual architecture, and fleshing out the earlier components and use cases. The conceptual architecture previously developed to outline the architecture of the Bitcoin core served as a baseline to explore the detailed concrete architecture. This architecture was improved upon, after the initial report, to move from a layered architecture style to a publish-subscribe model. The concrete architecture, however, moves beyond developing a theoretical framework for architecture based on research and documentation. Instead, the Understand software was used to critically examine existing components and dependencies along with discovering elements missing from the original conceptual architecture. In addition, this report further delves into the storage subsystem as the chosen level-two subsystem of Bitcoin Core. This includes breaking the storage subsystem down into sub-components, such as the Blockchain, Blocks, and other data. Once again, two use cases were included to demonstrate Bitcoin Core's key functionalities. Finally, the report concludes the lessons the group has learned, and the conclusion.

INTRODUCTION & OVERVIEW (ERIC)

The purpose of this report is to recover the concrete architecture of Bitcoin Core. This was done through the use of a code analysis tool called Understand. This tool allows for the mapping of entities to subsystems and creates a graphical view of the dependencies, which was analyzed thoroughly for our group to create the concrete architecture and perform a reflexion analysis. The first section of the report goes into detail about the derivation process for updating our conceptual architecture, recovering the concrete architecture, recovering a second-level subsystem concrete architecture, and performing reflexion analysis on both the top-level and secondary-level concrete architectures. The following sections go into detail on the modifications made to our conceptual architecture and why these updates were appropriate. Following that is a breakdown of our concrete architecture, including details about the subsystems, the subsystems of our second-level subsystem, and a reflexion analysis that analyzes the discrepancies between the conceptual and concrete architectures. The final section goes into the use cases of Bitcoin Core based on the recovered concrete architecture. Each Use Case covers a sequence diagram of that use case and a box-and-arrows diagram.

DERIVATION PROCESS (ERIC)

Through studying the source code of Bitcoin Core [3] and performing individual research, we were able to confirm our original conjectures on the conceptual architecture, while also gaining new knowledge and insight into the interacting components and architecture style. Using the new information we obtained through studying the source code, we were able to make modifications to our conceptual architecture, which involved the adding/removing of dependencies and modules and changing the architecture style from the layered style to a publish-subscribe style.

We derived our concrete architecture by using the code analysis tool Understand the Bitcoin Core source code. Each group member made their dependency graph of the system using Understand based on our newly updated conceptual architecture. When developing our

dependency graphs, in order for us to appropriately map source code directories and files to their subsystems, it was required that we did further research to understand how each subsystem functions and communicates. This included a thorough analysis of the source code files, and research on the different directory purposes in the source code (such as *bench*, *secp256k1*, *IPC*, *consensus*, etc.). Once this was done, we discussed our graphs and, using the Understand tool, formulated a refined dependency graph of the Bitcoin Core source code based on recurring patterns and logical reasoning (**Figure 1**). The dependency graph helped us see how each module truly interacted with each other, allowing us to create our concrete architecture (**Figure 3**) and perform a reflexion analysis, where we took note of all dependency differences between the new dependency graph and our conceptual architecture.

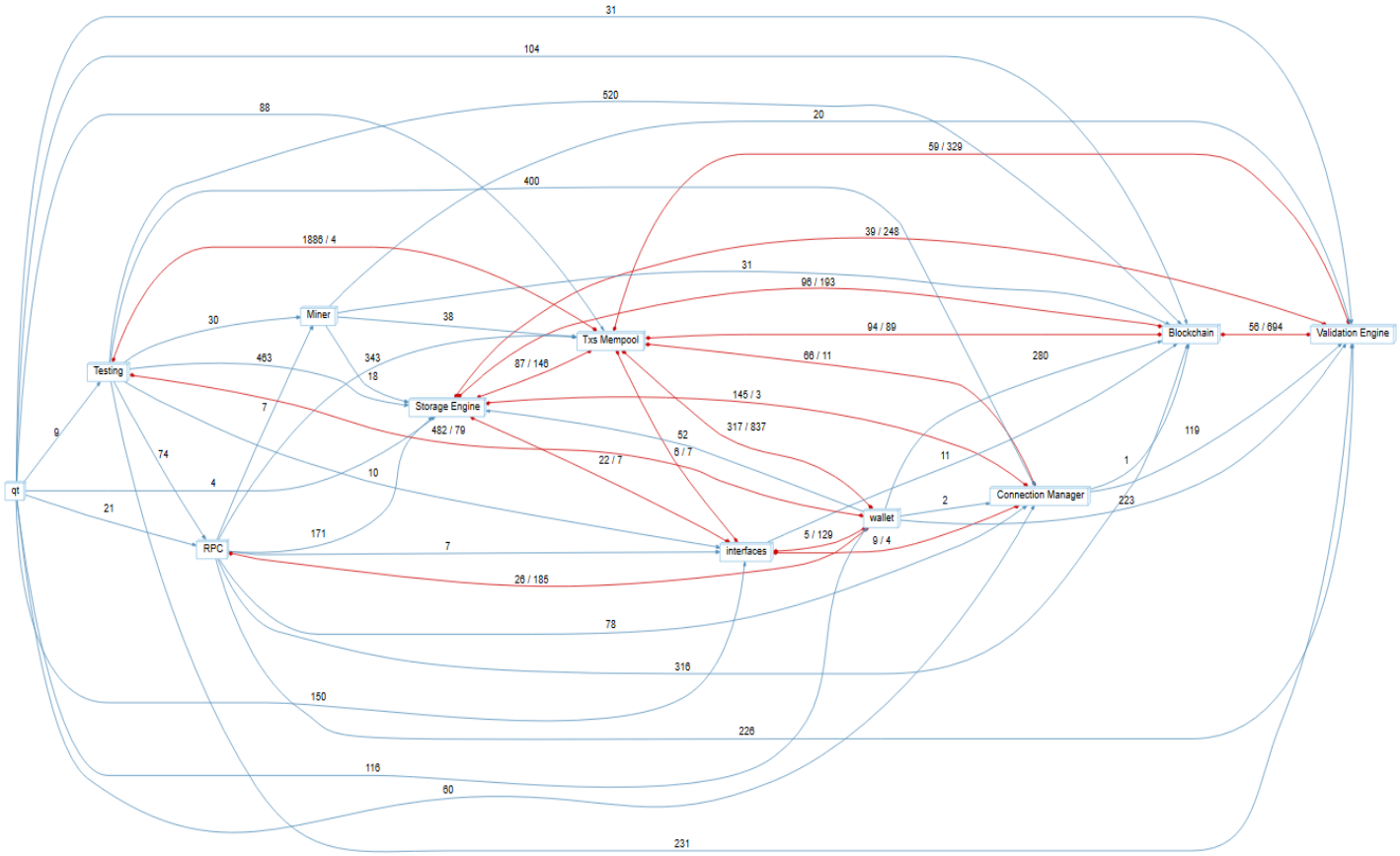


Figure 1. Understand Dependency Graph

When selecting our second-level subsystem to delve into, we chose to talk about the components related to the storage of Blockchain and other data. This includes the Blockchain and Storage Engine subsystems. We selected this due to its many interactions with other top-level components and the numerous, critical, and thoroughly defined subsystems. A similar process was used to derive the architecture of the subsystems. Since we did not have any conceptual architectures for our subsystems going into this report, each group member was assigned a couple of subsystems to formulate a conceptual architecture for. This involved extended research and analysis of the source code. The concrete architecture was then derived similarly using Understand, followed by a reflexion analysis of our chosen second-level subsystem.

CONCEPTUAL ARCHITECTURE - UPDATED

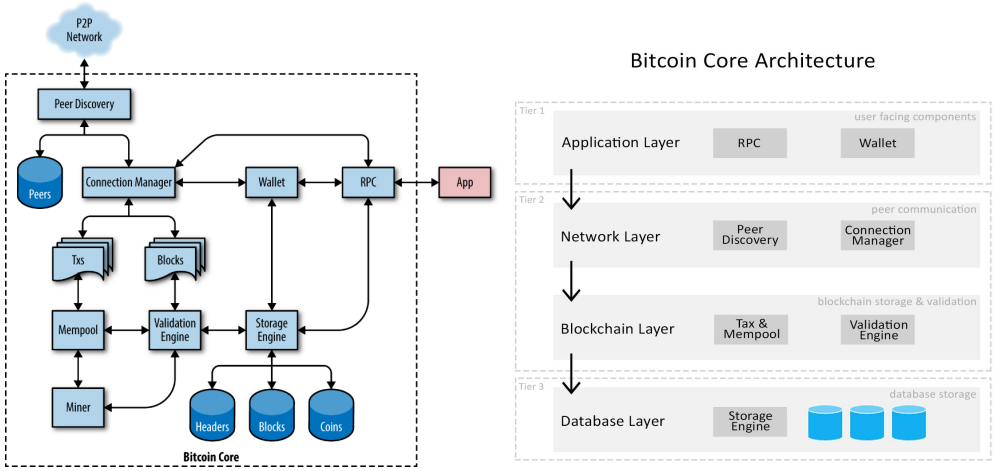


Fig. 2 Original Bitcoin Core Architectures

Originally, we studied 10 subsystems of Bitcoin Core and concluded that the main architectural styles of Bitcoin Core were a combination of peer-to-peer and layered. After receiving feedback on our original report, reviewing our previous report, and conducting further research on Bitcoin Core's source code on GitHub [3], we made appropriate modifications to our selected subsystems and architecture style. Our first modification to our subsystems was the removal of peer discovery as a top-level subsystem. After examining the functionalities of the subsystems, we concluded that peer discovery would be better suited as a second-level subsystem under Connection Manager. From our previous report, Connection Manager is responsible for the management of interactions between peer nodes, a very broad topic, while peer discovery was responsible for simply discovering potential peers through their IP address ports. Seeing as Peer Discovery is a very narrow scope in functionality, but is still associated with Peer Node functionality, we concluded it was appropriate to put Peer Discovery as a subsystem of Connection Manager.

Our second modification came from reviewing our A1 report and the documentation of the components. When reviewing this information, we discovered there was a dependency that we mentioned in our report but failed to add to the conceptual architecture. In A1, we mention the Wallet contains pairs of keys that are used in the signing of transactions, so a dependency between Wallet and Txs & Mempool is required.

Finally, we decided that a change from the Layered architecture style to Publish-Subscriber Style was necessary, and more appropriately suits the architecture of Bitcoin Core. The whole system involves a loosely-coupled collection of components, where each component carries out some operation that may trigger the operations of other components, as opposed to simply carrying out its own operations and finishing them.

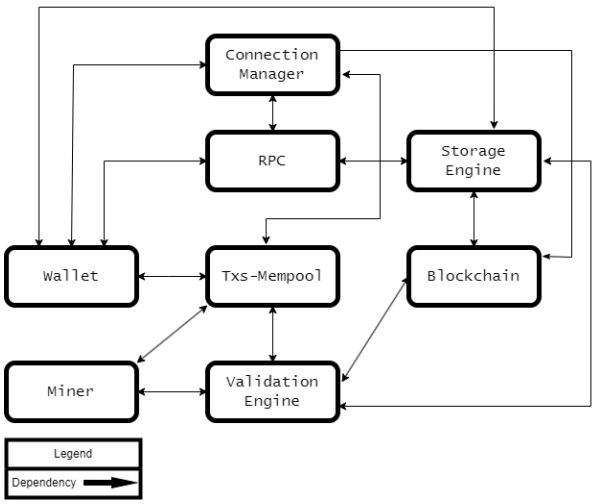


Figure 3. Updated Conceptual Architecture

CONCRETE ARCHITECTURE

Overview (Amy)

The concrete architecture of Bitcoin Core as outlined in Figure 3. was derived from both modified conceptual architecture and the Understand component dependency map. The above architecture identifies key dependencies and components which were missed in the conceptual architecture. The architectural style of the architecture is publisher-subscribe, where components broadcast messages that subscriber components can receive.

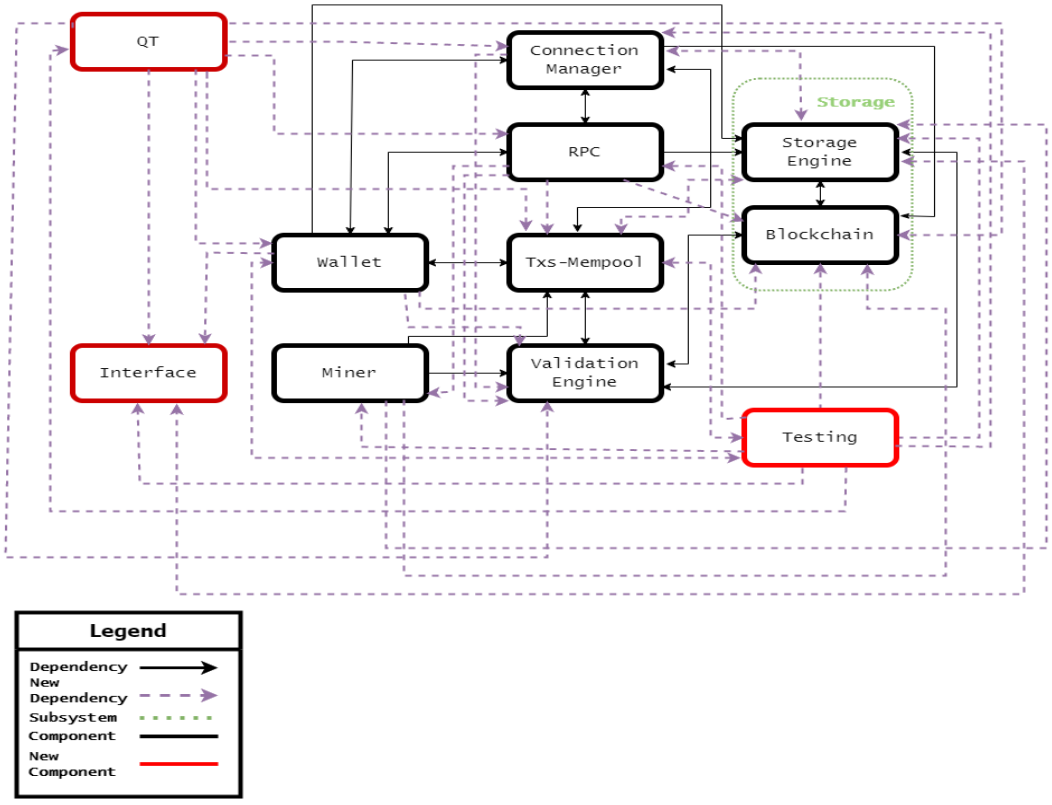


Figure 4. Concrete Architecture

Connection Manager

The connection manager is Bitcoin Core's implementation to manage interactions between peer nodes. This component is essential to the functionality of all components which require interactions with peer nodes, as the primary message thread in the connection managers is responsible for managing the queue of inbound messages from peer nodes, and for sending outbound messages toward various sockets. In terms of Implicit Invocation, the connection manager acts as a middleman between nodes that require communication with each other. Nodes send a request to the connection manager, which then forwards the request to the applicable nodes. This way, nodes do not require a direct connection with each other which maintains a steady flow of network traffic.

Miner

Mining is the process in which blocks are added to the blockchain. By adding blocks to the blockchain, a block reward and transaction fees are paid out. In terms of architecture, the miner is a component that acts on its own and does things by itself. Messages and information sent by this component must not be altered in any way as it is the most crucial in terms of what creates a monetary value for this whole system. It'll be required that this is consistent 99.999999 percent of the time. There is no scalability for this mining system. Security in terms of integrity, encryption, and non-repudiation is incredibly important. The new architecture of Implicit Invocation and Peer to Peer affects the Miner in the way that it will depend on more components, specifically the Blockchain, Txs Mempool, and the Storage Engine.

Storage Engine

Bitcoin Core uses a "full node" architecture to maintain a complete copy of the blockchain and validate and verify all transactions. The storage engine in Bitcoin Core manages this copy of the blockchain using a database management system called LevelDB, which allows for efficient retrieval and storage of data. When a new block is added to the blockchain, the storage engine indexes and stores the block data using LevelDB. Additionally, the storage engine includes features such as caching and compression to optimize the performance and efficiency of blockchain storage.

Validation Engine

The validation engine in Bitcoin Core verifies new transactions or blocks that are received against protocol rules to ensure their validity and rejects ones that have any errors or violations to prevent invalid entries. The validation engine also participates in the consensus process, which requires nodes to agree on the blockchain's current state, ensuring agreement on the validity of transactions and blocks.

Txs & Mempool

Txs are the most important part of the bitcoin system. They are data structures that encode the transfer of value between participants in the bitcoin system. Everything else in bitcoin is designed to ensure that txs can be created, propagated on the network, validated, and finally added to the blockchain. Each txs is a public entry in bitcoin's blockchain. Bitcoin txs create outputs. Outputs are recorded on the ledge, and most of these outputs create spendable chunks of bitcoin called UTXO. Txs & Mempool subsystem has a bidirectional dependency on Blockchain, Storage Engine, Validation Manager, Wallet, Connection Manager, and Testing. The Storage Engine is responsible for the storage of the validated transaction that is added to the block.

Additionally, a two-way dependency on Connection Manager as it manages the interaction, which includes Bitcoin transactions.

Wallet

After reflexion analysis, we discovered that the wallet interacts differently throughout the whole system when it uses the new implicit invocation architecture style. The wallet acts as a client to the nodes in the network as it sends requests and receives responses through the connection manager which, as mentioned recently, is the middleman between nodes. The wallet sends its private keys to the connection manager to be broadcasted to the other nodes. The wallet could use RPCs to request and receive information about the user's wallet through broadcasts through the connection manager.

Blockchain

Bitcoin Core will require the blockchain to be validated when a new node is created, which utilizes a headers-first approach to downloading the blockchain. The new node downloads the header of each block in the current longest blockchain and then downloads the remaining body of each block through concurrent connections with several peers. This blockchain is saved on the map "chainActive", containing the longest verified blockchain. This subsystem has bidirectional dependencies with Validation Engine, Txs & Mempool, and Storage Engine subsystems. This subsystem depends on Validation Manager for the validation of the legitimacy as a temporary situation, fork [9], where a block having more than one child has to be resolved. The Storage Engine will manage the storage of the blockchain itself. Txs & Mempool will be depended on to add newly validated blocks as well as transactions.

Testing

The Testing component is one of the new subsystems included in the updated architecture. This component includes experimental new features and augmentations which have yet to be approved for official usage, along with unit tests and automated testing tools for existing features. A Testing subsystem ensures that new releases of Bitcoin Core have minimal integration issues between different developers and that the code is functional. The component includes further developer tools to aid in software testing. This includes the TestNet component, a testing network where developers can acquire Satoshis that have no value and fewer functional restrictions exist (e.g. lessened validation for transactions) compared to the main network. Another such tool is Regression Test Mode, where developers may generate a pseudo-blockchain without other peers for testing purposes.

Qt

The Qt component is responsible for all code that is related to the graphical user interface. It contains two main classes: "BitcoinCore" and "BitcoinApplication". The first one encapsulates the startup and the shutdown logic and also allows running startup and shutdown in a different thread from the UI thread. BitcoinApplication extends QApplication and is the main Bitcoin Core application object [6]. This component is dependent on the Wallet, RPC, Connection Manager, Txs & Mempool, Storage Engine, and Interface components in order to display a graphical interface for Bitcoin Core users.

Interface

The Interface component simply provides a common interface for the other components of Bitcoin Core to interact with each other. Interfaces that are defined within the Interface

component include Chain (used by the Wallet to access the blockchain and mempool state), Node (used by the GUI to start and stop Chain clients), and Wallet (used by the GUI to access wallets) [6].

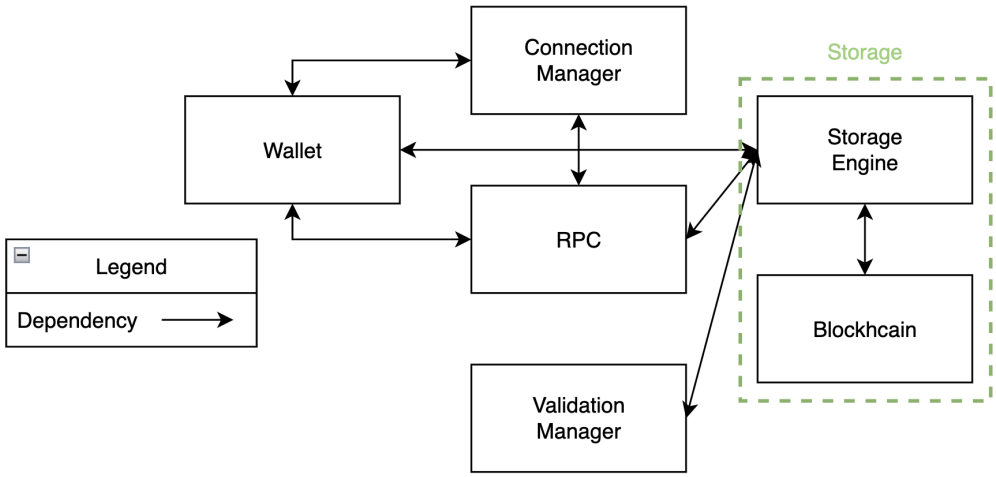
RPC

An RPC is short for Remote Procedure Call. This means that a program is calling for a procedure to execute and run in a different address space. This RPC allows for an easier runtime for the programmer as they do not need to include details of the remote occurrence. It is as simple as a client calling the server for information through the entire network, the RPC is the actual call. The RPC interacts with almost every component, including Storage engine, Connection manager, Wallet, Miner, Txs & Mempool, Blockchain, Testing, and Validation engine. It is key that all components use RPCs as a form of communication.

SECOND-LEVEL SUBSYSTEM (STORAGE OF BLOCKCHAIN + OTHER DATA)

Overview

Conceptual View

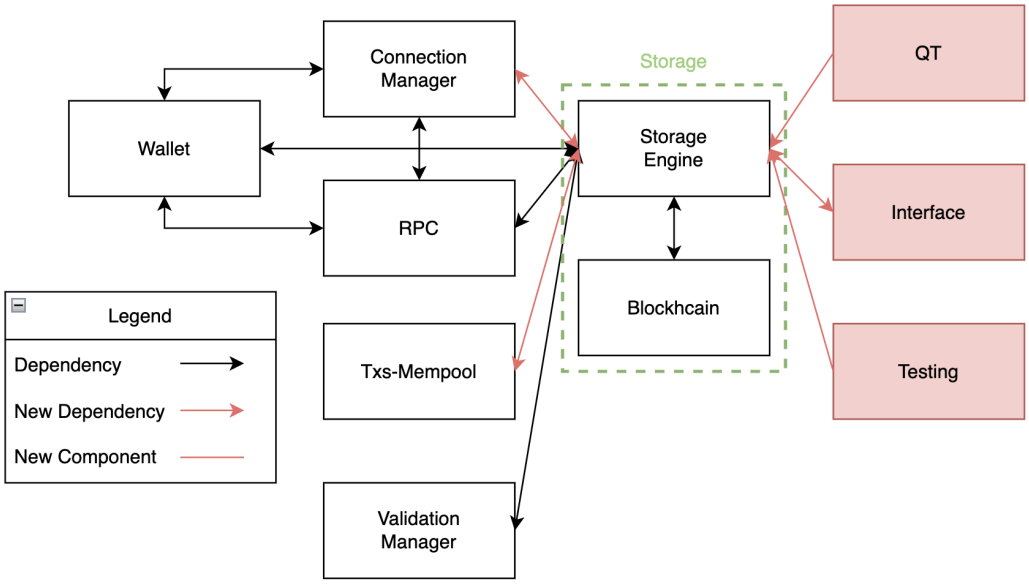


Main Style: Peer to Peer

The subsystem Storage of Blockchain and other data uses the Peer to Peer style due to the distribution of data around the network of nodes. The style is based on the data of each node knowing the other data of the other nodes and with relevance to the blockchain. There is no central database and transactions are verified by each node using consensus. Each node, through LevelDB and the Cache, has a copy of the entire database, which are mainly the transactions and addresses.

Observing the Storage of Blockchain/Other Data subsystems, we will show both the conceptual alongside concrete architecture of the system. We know several components make up the entire storage. The main components are the blocks themselves, the chain states attached to the block, the coins which represent the medium of value relating to the blockchain, the cache that stores the blockchain's unconfirmed transactions, LevelDB management system is what stores blockchain information in the key pairs, and consensus makes sure the entire network of nodes is in agreeance of the state of a node.

Concrete View



The concrete architecture for the storage of blockchain and other data is still modeled in the Peer to Peer style. The blocks are within the blockchain and act as the implementation for what makes up the blockchain in order to be stored, and what data is in the blocks that are important for storage. The coins, cache, chainstate, levelDB, and consensus all act upon the entire blockchain of blocks. The coins, cache, levelDB, and consensus deal directly with the transaction of the blockchain while the chainstate deals more with the blocks and the blockchain.

Blocks

We see that each Blockchain has a Block Storage, each block points to the other with information that is stored. A block contains a BlockHash, previousBlockHash, version, time, nonce, nBits, Merkle root, and stored transactions. Putting the information together creates the blockchain. The blockchain records all transactions, referenced in chainstate, with each block containing its own collection of transactions.

The blockchain is not stored traditionally in one singular place, but within the storage engine, which is throughout the entire network.

Interactions with other components include relation to chainstate, storage in the blockchain, cache storage of blockchain, and consensus when verifying blocks.

Chainstate

Chainstate differentiates itself from the directory of blocks in that it's not the storage of the actual block, but the state of the latest block which includes every spendable coin, the owner, and its value.

The main interaction of chainstate and another component is the state of the blockchain will need to call upon the consensus mechanism in order to verify the state of a blockchain or if anything the state is about to be updated.

Coins
Coins are a single unit of currency that is associated with the unit of a Bitcoin. The coins are what create the value of the currency itself in the market and work as an exchange or a storage of value. The bitcoin network does not recognize or store the coins themselves but records the transactions in terms of how much bitcoin is made going in and out of addresses related to the blockchain. [1]

Cache
The Cache is the high-speed data storage level that is needed to store a subset of data and to speed up data processing. UTXOS are indivisible pieces of Bitcoin currency locked to a specific owner that are recorded in the blockchain. The UTXO represents all unspent transaction outputs and validates new transactions.

The mempool does act as a version of the cache, which contains all unconfirmed transactions of the node and has not been stored in the block yet.

LevelDB
LevelDB is what is used by Bitcoin to save the transaction index and transaction hashes. As mentioned earlier, each block stores all transactions in relation to its history of itself. This is also where UTXOs are stored in the block. It stores key-value pairs for the blockchain data.

Consensus
Consensus is what encapsulates our security and trust agreements when mining blocks, creating new blockchains, and doing a lot of security checks for validation. As mentioned in the top-level conceptual architecture, there are proof-of-works and proof-of-stake. It is the process where the nodes agree on the state of the blockchain. It maintains that all nodes in the network have the same view of the blockchain.

REFLEXION ANALYSIS

Top-level

Storage Engine ↔ Txs & Mempool
Initially, we had no dependencies between Txs & Mempool and Storage Engine in our conceptual architecture. The rationale for why they have this bidirectional relationship is through blocks. Blocks are stored in the Storage Engine under the block database. They are a collection of validated transactions that are recorded on the Blockchain [8]. However, a transaction is considered verified only after the block is it associated with has been added to the blockchain. Due to the structure of the blockchain and blocks, as more blocks are added after a certain block, the transactions contained in that block become more secure and stable [9].

Wallet → Validation Engine
Initially, we had no dependencies between Wallet and Validation Engine, however, it appears that the Wallet depends on the Validation Engine. By examining the source code and the dependency graphs in Understand, Wallet depends heavily on a subsystem of the Validation Engine, called Policy, Policy is responsible for making logical assessments about transactions and for doing fee estimation [6]. One of the Wallet's responsibilities is the creating and signing of transactions. The

Wallet calls upon the Policy component to implement the proper logical assessments to validate that the transaction is verified to be signed by a user using the keys in their wallet [7].

Storage Engine ↔ Connection Manager
Initially, we had no dependencies between the connection manager and the storage engine, however, it appears that there is a bidirectional dependency relationship between the two components. The storage engine depends on the connection manager through transactions. The connection manager is responsible for managing interactions between peer nodes. Peer nodes can request data from each other, such as making a transaction between two nodes [8]. Transaction information is stored in blocks, which can only be distributed through the connection manager, thus the rationale for the storage engine's dependency on the connection manager. A Bitcoin client maintains a UTXO database/pool, a set of all UTXO of the blockchain that is housed as an indexed database in persistent storage [4]. When peer nodes have collections of unspent output, the connection manager requires the use of the UTXO pool as a storage area.

Connection Manager → Validation Engine
Initially, there were no dependencies between the connection manager and the validation engine, however, there appears to be a dependency from the Connection Manager to the Validation Engine. The Connection Manager is responsible for receiving incoming Blocks from the Peer Network. These newly acquired blocks will be stored as unvalidated initially. As such, The Connection Manager depends on the Validation Engine to ensure the validity of incoming blocks, such that they may be connected to the legitimate Blockchain [4].

Miner, Wallet → Storage Engine
Initially, the Miner had no dependencies on the Storage Engine, and the Wallet had a bidirectional dependency relationship with the Storage Engine. After the construction of the concrete architecture, we see that Storage Engine does not depend on the Wallet, but the Wallet still depends on it. This is because, as mentioned previously the Storage Engine simply acts as a storage of transaction information for the Wallet to use, thus actions involving the Wallet depend on the Wallet's initiative. We also see a dependency between the Miner and Storage Engine, which was not covered in the conceptual architecture. By examining the dependency diagrams through Understand, we see a major component the Miner depends on is hashing functions related to CBlockHeaders. This correlates to the responsibilities of the Miner, which generates the hash and header for a block using the "getblocktemplate" RPC.

RPC → Miner, Txs & Mempool, Blockchain, Validation Engine, Storage Engine
In the initial conceptual architecture, RPC had bi-directional dependencies with the wallet, connection manager and storage engine. However, there appears to also be one-way dependencies from the RPC to the Miner, Transactions/Mempool, Blockchain, and Validation Engine. Given the wide array of services that can involve remote access, the RPC module depends on almost every component in the Bitcoin Core architecture. This is because while RPC servers allow nodes to request services from peers, some of the actual functionalities are implemented using existing components in the codebase [6]. For instance, though a user may complete a transaction through an RPC server, the details and functionalities of the transaction exist locally.

Miner → Txs & Mempool (not ↔)

Initially, it was theorized that a bi-directional dependency existed between the Miner and the Transactions/Mempool. Since Miners are responsible for mining blocks, this includes adding unverified to the memory pool to await validation. This is responsible for creating the dependency of the Miner on Txs&Mempool. However, this dependency is not bi-directional like expected, because after a block is sent to the Mempool, it will be validated before being added directly to the Blockchain; it will not be sent back to the Miner.

Miner → Validation Engine (not ↔)
Initially, it was theorized that a bi-directional dependency existed between the Miner and the Validation Engine. Since Miners are responsible for generating mining blocks, it requires a method to ensure the newly created Bitcoin is valid, for this reason, there exists a one-way dependency from the Miner to the Validation Engine, to ensure Consensus rules are followed. However, the Validation Engine does not rely on the Miner. This is because the validation engine specifically exists to check whether blocks can be trusted, regardless of their Miner or origin [2].

Wallet, Miner → Blockchain
Initially, neither the Wallet nor the Miner components had dependencies on the Blockchain directly. However, the updated concrete architecture shows that both components are dependent on the Blockchain. The Miner is responsible for the creation of new blocks, therefore, the Miner depends on the Blockchain component to have the new block processed in the Blockchain. The Blockchain handles the passing and submission of the newly mined block to the validation engine, after which the block is added to the Blockchain(if it is valid), thus completing the creation of a new block [2]. Likewise, the Wallet depends on the Blockchain because the wallet is a client to the nodes in the network, therefore, it must occasionally access the Blockchain to ensure the local Blockchain is sufficiently updated. The Wallet also must scan the Blockchain for Transactions that are directed to or from the user [5].

2nd level subsystem

The storage of the blockchain is a key aspect of the bitcoin core architecture. The subsystem consists of many different components utilizing blocks, chainstate, coins, cache, LevelDB, and a consensus. After comparing the two architectures, a key difference we noticed was the dependencies connected along with the storage of the blockchain. Within the concrete architecture, the blockchain is closely tied with the storage engine under its own subsystem rather than different layers of a layered architecture style. With this new architectural style, many new dependencies are created working towards the storage of the blockchain. Initially, many of the components did not have direct dependencies with the storage engine. The Txs and Mempool, the Miner, and a Testing component now have dependencies with the storage subsystem.

Moreover, new components of the subsystem are found that are not apparent in the conceptual architecture. The chainstate is a component that keeps the current state of the blockchain up to date, this includes things such as the current balance of each Bitcoin address. The information is taken from the blockchain and is used to validate new transactions and blocks. The LevelDB is a database that is the main storage of blocks and the blockchain. Each block contains a list of transactions that are confirmed by the bitcoin network and the validation engine. Blocks also include data such as the block's hash, timestamp, and the hash of the previous block in the chain. Coins are a representation of unspent currency (UTXOs) in the Bitcoin network. Each UTXO represents a specific amount of Bitcoin that has not yet been spent. The cache component in the Bitcoin Core storage subsystem is used to speed up access to

frequently accessed data, such as UTXOs and blocks. The consensus is what encapsulates our security and trust agreements when mining blocks, creating new blockchains, and doing a lot of security checks for validation.

USE CASES

Use case #1 - New User making a payment at a retail store

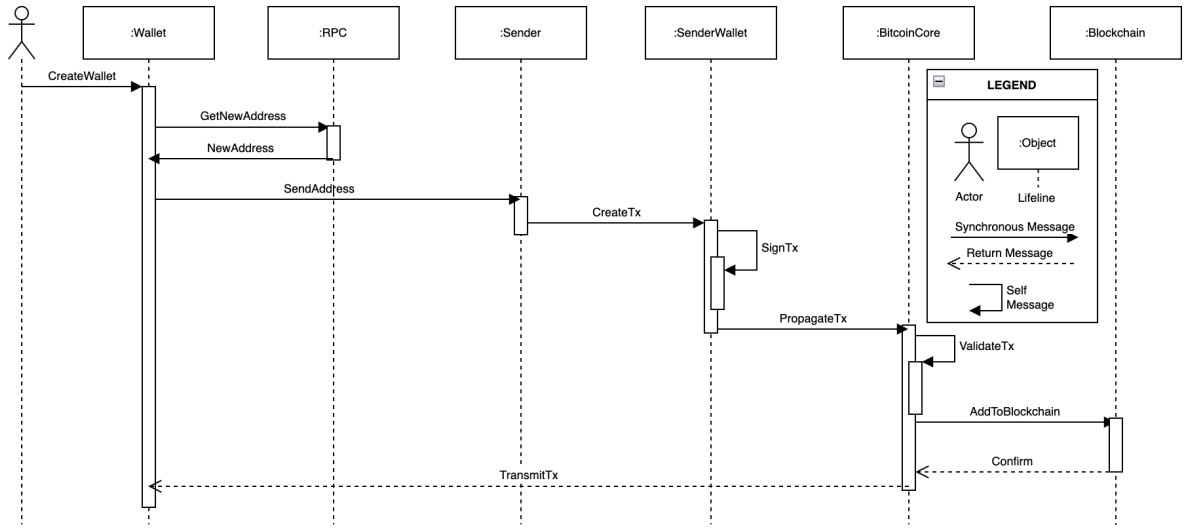


Figure 5. Sequence Diagram of Use Case #1

When a new user downloads and runs a mobile wallet application, the application automatically creates a wallet comprising addresses and keys. However, these addresses will only become known in the network after being referenced by another sender. If someone wishes to send bitcoin to this user, they must construct and sign a new transaction using their private keys. The transaction is then propagated and validated through well-connected nodes in the network before being listened to by the recipient's wallet that holds the corresponding address. At this point, the transaction is unconfirmed since it has not yet been recorded in the blockchain. Once the transaction is included in a block and added to the blockchain, which occurs every 10 minutes, it becomes finalized. This use case is described as a sequence diagram in figure 5.

Use case #2 - How new transaction gets verified

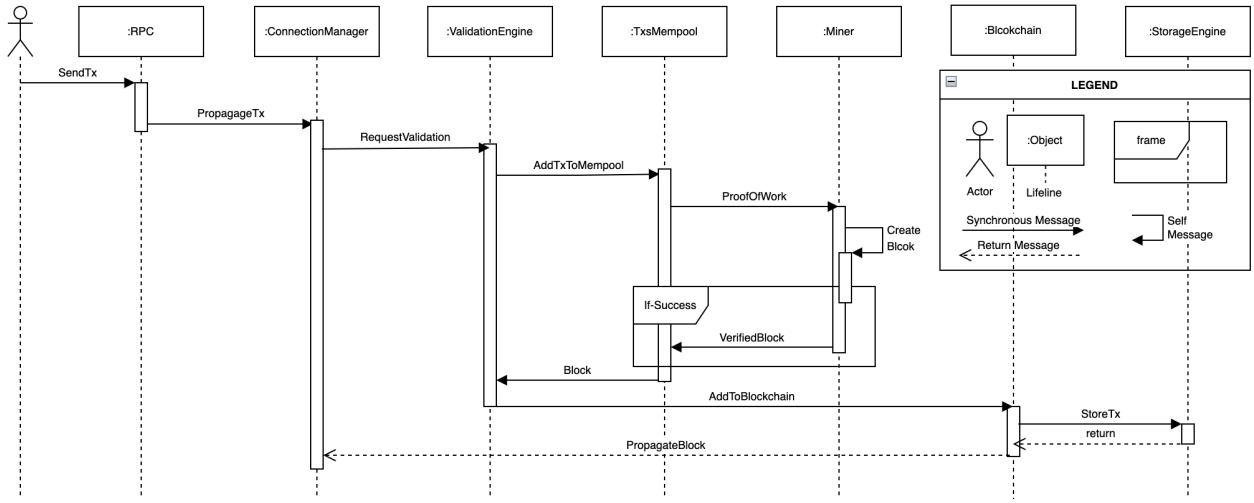


Figure 6. Sequence Diagram of Use Case #2

Recalling Use Case #1, which primarily focused on user interaction with the application, Use Case #2 delves into the implicated details of the Bitcoin Core application. Suppose the sender from Use Case #1 initiates a new transaction, and the Wallet forwards it to the RPC subsystem. To begin with, the Connection Manager transmits the transaction to the Bitcoin network to include it in the blockchain. Next, the Validation Manager and Txs&Mempool receive the new transaction. When the network nodes observe the transaction, the Miner generates a candidate block and attempts to verify the unverified transaction using Proof-of-Work computation. Upon successful validation, the Miner announces the winning block to the network, and the new transaction becomes a part of the Blockchain and the Storage Engine. This process is described in a sequence diagram in figure 6.

NAMING CONVENTIONS

UTXO - Unspent Transaction Output
Txs - Transaction

CONCLUSIONS

In conclusion, our report has provided a detailed analysis of the concrete architecture of Bitcoin Core. The Understand software was used to perform a reflexion analysis on the top level, leading to the discovery of missing elements and dependencies. An in-depth analysis was also done on the storage subsystem which was determined to be a Peer to Peer architecture based on the fact that each node knew the data of the other nodes and is composed of many of its own sub-components. The report recommends a switch from the Layered architecture style to a Publish-Subscriber model, which better aligns with the loosely-coupled nature of Bitcoin Core. The report also presents two use cases to illustrate Bitcoin Core's key functionalities, such as how a transaction becomes confirmed and added to the blockchain. Overall, our report provides a comprehensive analysis of the concrete architecture of Bitcoin Core, revealing new insights into

the subsystems and their interdependencies and highlighting the importance of rigorous analysis and documentation in software architecture design. We hope that this report serves as a valuable resource for those seeking to understand the inner workings of Bitcoin Core's architecture and its place in the broader landscape of blockchain technology.

LESSONS LEARNED

The aim of this report is to outline the concrete architecture of Bitcoin Core and the deeper architecture of one subsystem. We ran into some difficulties while doing this. One major difficulty we encountered was the process of translating the system components and dependencies as described by Understand into the concrete architecture. It was difficult to know which dependencies were most relevant, and when to add a new component to the diagram. To alleviate this difficulty, the team further researched the various components of Bitcoin Core from its GitHub documentation. This furthered our understanding of the system and aided us in creating the architecture. Additionally, in the future, we would better organize the structure of the report. Near the report's completion, it was evident that the content would exceed the assignment's page limit, due to repetitive content from the first assignment and between components written by various team members. Better organization would prevent this problem in the future.

REFERENCES

[1] Binance Academy. (n.d.). *Coin*. Binance Academy. Retrieved March 23, 2023, from https://academy.binance.com/en/glossary/coin

[2] *Bitcoin Core Validation*. Validation - Bitcoin Core Features. (n.d.). Retrieved March 23, 2023, from https://bitcoin.org/en/bitcoin-core/features/validation

[3] Bitcoin. (n.d.). *Bitcoin/SRC at master · Bitcoin/Bitcoin*. GitHub. Retrieved March 23, 2023, from https://github.com/bitcoin/bitcoin/tree/master/src

[4] Bitcoin. (n.d.). *Bitcoin/validationinterface.h at 4b5659c6b115315c9fd2902b4edd4b960a5e066e · Bitcoin/Bitcoin*. GitHub. Retrieved March 23, 2023, from https://github.com/bitcoin/bitcoin/blob/4b5659c6b115315c9fd2902b4edd4b960a5e066e/src/validationinterface.h#L78

[5] Bitcoin. (n.d.). *Bitcoin/wallet.cpp at 4b5659c6b115315c9fd2902b4edd4b960a5e066e · Bitcoin/Bitcoin*. GitHub. Retrieved March 23, 2023, from https://github.com/bitcoin/bitcoin/blob/4b5659c6b115315c9fd2902b4edd4b960a5e066e/src/wallet/wallet.cpp#L4514[10

[6] Chaincodelabs. (n.d.). *Bitcoin-core-onboarding/1.1_regions.ASCIIDOC at main · chaincodelabs/bitcoin-core-onboarding*. GitHub. Retrieved March 23, 2023, from https://github.com/chaincodelabs/bitcoin-core-onboarding/blob/main/1.1_regions.asciidoc

[7] *Chapter 5: 'wallets'*. Chapter 5: 'Wallets' · GitBook. (n.d.). Retrieved March 23, 2023, from https://cypherpunks-core.github.io/bitcoinbook/ch05.html

[8] *Chapter 6: 'transactions'*. Chapter 6: 'Transactions' · GitBook. (n.d.). Retrieved March 23, 2023, from https://cypherpunks-core.github.io/bitcoinbook/ch06.html

[9] *Chapter 9: 'the Blockchain'*. Chapter 9: 'The Blockchain' · GitBook. (n.d.). Retrieved March 23, 2023, from https://cypherpunks-core.github.io/bitcoinbook/ch09.html

[10] *TheBitcoin application itself forwardinbitcoinimprovementproposals ... - gi*. (n.d.). Retrieved March 23, 2023, from https://dl.gi.de/bitstream/handle/20.500.12116/16570/DFN-Forum-Proceedings-001.pdf