Concrete Architecture Analysis for Bitcoin Core

by
Gabriel Lemieux (19gml2@queensu.ca),
Jack Taylor (17jmt5@queensu.ca),
Jack Fallows (19jef2@queensu.ca),
Jiacheng Liu (19jl193@queensu.ca),
Maninderpal Badhan (19msb14@queensu.ca),
Nil Shah (20ns3@queensu.ca)
Mar. 23, 2023

Abstract:

This report presents an in depth analysis of the concrete architecture of Bitcoin Core. By utilising the SciTools software Understand, we were able to recover the concrete architecture of Bitcoin Core and identify discrepancies between it and our proposed conceptual architecture. We highlight key findings and justifications for the presence of certain divergences, such as frequently used files with no clear component. We also perform sub-component analysis on the Connection Manager, We outline our lessons learned throughout the process and include our recommendations for others who attempt to recover the concrete architecture of a large software system.

Introduction and Overview:

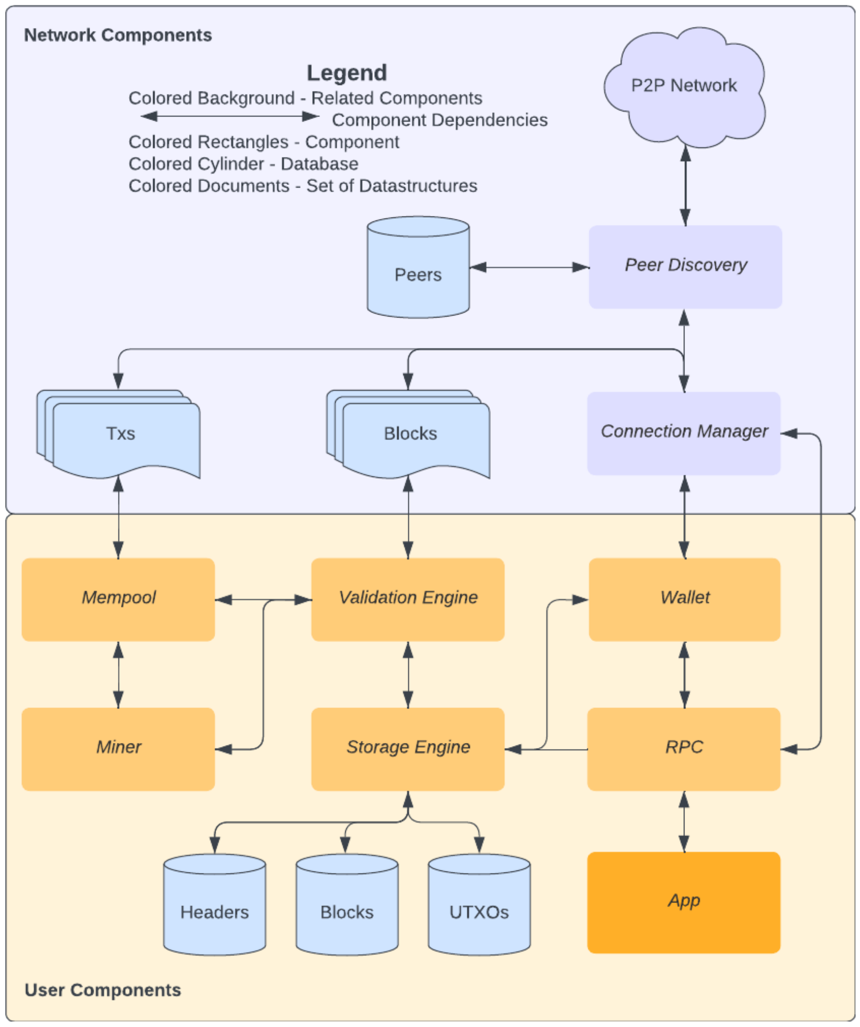
In contrast to the conceptual architecture of a software system, which is developed based on an idealistic and simplified overview of the software being developed, a system's concrete architecture represents the actual implementation of the technology. If a conceptual architecture is analogous to a blueprint of a house, for example, the concrete architecture is the actual house as it is built.

The derivation process for a concrete architecture is significantly more involved than that of a conceptual architecture; it requires access to the source code and substantial dependency analysis. By using software such as *Understand* to graph dependencies and other code relationships, it becomes possible to construct a diagram that represents the concrete architecture of a software system.

The aim of this report is to compare the derived concrete architecture with the conceptual architecture. With the conceptual architecture providing a view of the top-level components working within the system, the concrete architecture provides a view of the actual implementation, which often differs from the conceptual since required modules and functions are added to implement the required functionalities, causing additional dependencies and discrepancies with the conceptual architecture. These discrepancies will be analysed through reflexion analysis, examining the rationale for the differences in the concrete architecture. The use cases from the previous report were created using the concrete architecture, providing a more accurate and detailed execution of the use cases in the actual system.

Conceptual Architecture

Here is our original proposed conceptual architecture:

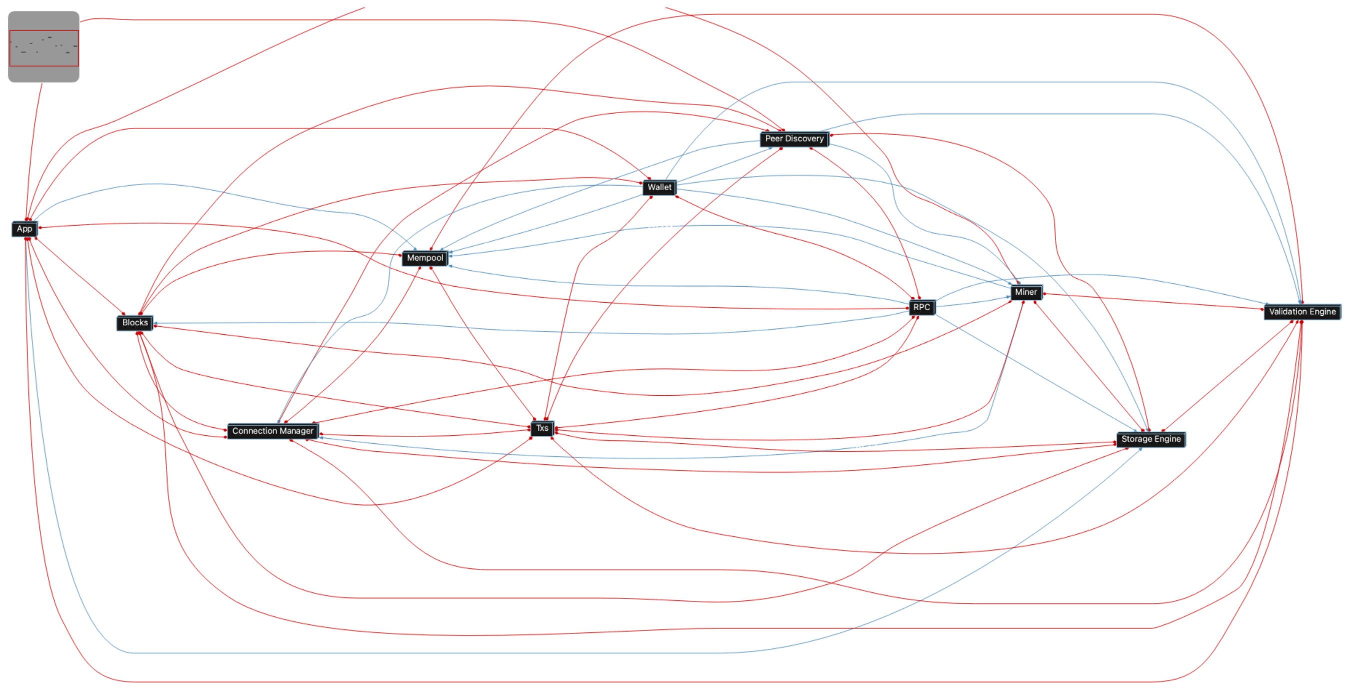


Derivation:

To be able to recover the concrete architecture for the Bitcoin Core system, we used the Scitools Understand platform. This tool allows users to analyse code repositories and create architectures based on that code. To generate the graphs we created an architecture for the system based on our conceptual architectures components, and then grouped the code files into the component we felt best fit its function. To determine which component a file or folder should belong to, we used the name of the file, the comments inside the file, and the code itself. After placing all the files inside our architecture, we were able to generate a dependency graph and recover the concrete architecture.

The dependency graph allows us to gain a better understanding of the dependencies between the components. Below is the dependency graph that was generated from the src file of

Bitcoin Core using the Understand tool. The graph is extremely complex and contains many dependencies. Explaining each dependency in depth would be unrealistic for a report of this scope; therefore, we will be analysing the most important divergences for each of the components from our conceptual architecture. It should also be noted that we did not sort all files that served a general purpose across the application, such as utilities.



We could not figure out how to make the components' font size bigger.
Here are the components from left to right: App, Blocks, Connection Manager, Mempool, Txs, Wallet, Peer Discovery, RPC, Miner, Storage Engine, Validation Engine

In order to determine which of the recovered dependencies were legitimate and which were due to missorting of files, we created the following matrix to organise our analysis efforts:

Architecture Dependencies

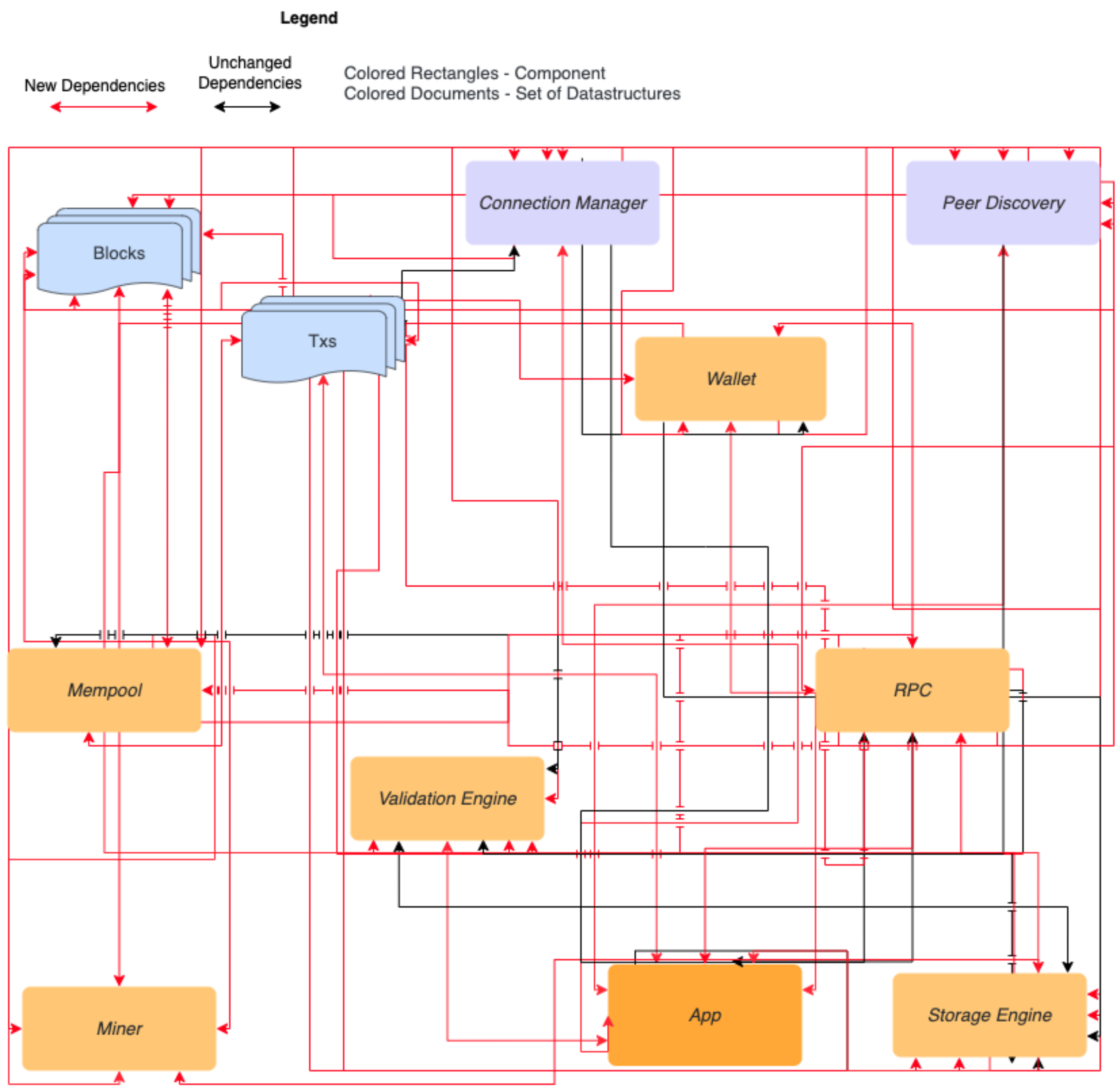
	App	Blocks	Storage Engine	Connection Manager	Miner	Txs	Wallet	Peer Discovery	RPC	Mempool	Validation Engine
App		100 / 55	18	42 / 16	3 / 14	139 / 2	127 / 127	93 / 72	54 / 6		87 / 20
Blocks	55 / 100		170 / 58	69 / 39	100 / 14	126 / 40	39 / 35	44 / 260		6 / 5	105 / 457
Storage Engine	18	58 / 170		8 / 32	19 / 14	60 / 76	17	35 / 9	47 / 9	4 / 2	35 / 197
Connection Manager	16 / 42	39 / 69	32 / 8			48 / 37	35	27 / 25	1 / 68	29 / 3	88 / 23
Miner	14 / 3	14 / 100	14 / 19			39 / 39	57		7	15	25 / 14
Txs	2 / 139	40 / 126	76 / 60	37 / 48	39 / 39		223 / 664	88 / 11	151 / 15	192 / 150	90 / 240
Wallet	127 / 127	35 / 39	17	35	57	223 / 664		3	21 / 226		246
Peer Discovery	72 / 93	260 / 44	9 / 35	25 / 27	57	11/ 88	3		12 / 25	3	149 / 2
RPC	6 / 54	116	9 / 47	68 / 1	7	15 / 151	226 / 21	25 / 12		24	83
Mempool		5 / 6		3 / 29	15	150 / 192		3	24		33 / 18
Validation Engine	20 / 87	457 / 105	197 / 35	23 / 88	14 / 23	240 / 90		165		18 / 33	

Present in Conceptual
Under Investigation
Clearly Justified
No Dependencies
Not Investigated

This matrix shows each component and its outgoing/incoming dependencies from each other component. We label all dependencies present in our original conceptual architecture in green, all dependencies being actively investigated in yellow, all dependencies with a clear justification in blue, no dependencies in light purple, and dependencies for which we have not provided a clear explanation in white.

Concrete Architecture

Here is our recovered concrete architecture



Conceptual vs Concrete Architecture Dependency Discrepancies

The following new dependencies were investigated after recovering the concrete architecture.

App -> Blocks

The file chain.h from the App component includes blockfilter.h, as the Apps blockchain class needs to implement a method that can check if blocks match using a filter. The purpose of the filter is to expedite the process of searching all blocks to find if there is a match. All other dependencies rely on the same inclusion of block filters, so this divergence is justified.

Wallet->PeerDiscovery

The file walletdb.cpp from the Wallet component needs protocol.h from the Peer Discovery component for information related to protocol standards with transactions. The Wallet init file init.cpp needs net.h for external signing from Peer Discovery.

PeerDiscovery->Mempool

There is only a single dependency from Peer Discovery to Mempool; the file net_processing.cpp includes mempool_entry.h. This is due to the Peer Discovery needing the entry point for the Mempool to store new transactions.

ValidationEngine<->Txs

There is a heavy dependence from the Validation Engine on Txs; for example, validation.h includes feerate.h and tx_verify.cpp includes coins.h. The reason for this is obvious; the Validation Engine handles validating transactions and blocks that include the transactions, and it requires lots of functionality and information from the Txs component to do this.

The Txs component has a dependency on the Validation Engine where transaction.cpp includes validation.h; this is because for a given transaction to be considered valid it needs to be verified that the block it came from is valid.

RPC->Miner

mining.cpp is a RPC mining helper class that requires the miner.h from the Miner component to function. RPC handles remote procedure calls which are required in the Miners operations. httprpc.cpp from RPC also includes hmac_sha256.h from Miner. The reason for this is explained below.

We have put all components related to hashing and the sha256 protocol in Miner, as we thought this was the best place for these files. By definition, mining is the process of solving the current block which involves finding a solution to the sha256 algorithm; however, it may cause many divergences as hashing and sha256 are present in many places across the software. Hashing would also make sense in the Blocks component but we felt it was a better fit in the Miner component; in general, it is required in many places.

RPC->Blocks

RPC has a strong dependency on Blocks; blockchain.cpp includes chain.h, mining.cpp includes chain.h, rawtransaction.cpp includes chain.h, and txoutproof.cpp includes chain.h to name a few. These are all RPC related files that need information related to current transactions and blocks and their validity so that a client can make calls to a server and receive a response.

Connection Manager<->Mempool

Mempool only has a few dependencies on Connection Manager in the form of client version checking to maintain a persistent mempool so that any client updates that affect the mempool structure can be taken into account; for example, mempool_persist.cpp includes clientversion.h. Connection Manager has a heavier dependence, but still minimal, on Mempool, as it needs to know the parameters related to mempool size, limits, and arguments.

Connection Manager<->Validation Engine

The Connection Manager depends on the Validation Engine because it needs to know that the current chain state and coins are valid before broadcasting this information to other nodes. This is in the form of chainstate.cpp includes validation.h, chainstate.h includes validation.h, chainstatemanager_args.cpp includes validation.h, chainstatemanager_args.h includes validation.h, coin.cpp includes validation.h, and context.cpp includes validation.h. There are some additional dependencies along a similar line of reasoning.

The Validation Engine depends on the Connection Manager for version-related info only, CheckTransaction Uses PROTOCOL_VERSION, and validation.h includes version.h are some of the main dependencies.

Mempool->Blocks

The main reason a dependency exists between Mempool and Blocks is that mempool_args.cpp includes chainparams.h. We put files related to the chain, made of blocks, in the Blocks component. The mempool is a place to store temporary but unconfirmed transactions that are to be added to the chain.

Blocks->Mempool

The connection is bidirectional yet we were unable to figure out what this dependency is. The explanation from Understand only shows how files in Mempool rely on Blocks. The fact that mempool_args.cpp includes chainparams.h explains part of the picture; Mempool needs to know parameters related to the block chain and block composition in order to correctly store and organize new transactions.

RPC->Mempool

The RPC component handles remote procedure calls. rpc_mempool.cpp includes mempool_entry.h, which means that the RPC file is specific to mempool remote procedure calls. The RPC likely makes calls to the mempool as it will handle adding new txs when they are received.

Wallet->Miner

The key.cpp file from Wallet includes hash.h from Miner. This is justified, as hashing a key is a common practice for the wallet to perform. As previously mentioned, hashing-related files are placed in Miner as we felt it was the best fit.

Txs<->Blocks

There are several files in Blocks that depend on Txs, block_assemble.cpp includes txmempool.h in addition to blockencodings.cpp including txmempool.h. This is fairly straightforward, the Blocks component needs information about the current txs in the mempool to construct a new block. There are also several files in Txs that depend on Blocks, txdb.cpp Includes chain.h and txmempool.cpp Includes chain.h. These are justified by txs requiring the ability to parse the chain and get past txs.

Wallet <-> Txs

The Txs component depends on the Wallet to receive the keys it needs to validate ownership of coins for a given transaction. The Wallet requires information about the fees for a transaction from the Txs Component (m_pay_tx_fee types CFeeRate at wallet.h).

Peer Discovery -> Validation Engine

Peer Discovery gets information about the active chain and its active state from the validation Engine. This information is used to get the last block that a caller has in the main chain. ProcessMessage calls ActiveChainstate at net_processing.cpp.

App <-> Wallet

The App needs the user's wallet information to be able to display it and allow the user to be able to interact with their wallet in addition to getting the addresses of the coins owned by the user as well. Wallet gains the information about changes done to the wallet from the user through interface.cpp.

Peer Discovery -> Miner

As previously explained, the Miner contains all the hashing related files. Peer Discovery requires hashing functionality to hash information sent to the network, and to unhash information received from the network. prepareForTransport calls Hash at net.cpp.

Storage Engine <-> Blocks

The Storage Engine handles storing all the blocks and uses BLOCK_VALID_TREE from chain.h to make sure all parent headers are found. This dependency makes sense since headers are a subsystem in the Storage Engine. Since blocks are stored in the Storage Engine, blocks require the location of specific blocks to know the current state of the rest of the chain. GetLocator types CBlockLocator at chain.cpp.

Peer Discovery / Txs -> RPC

Both Peer discovery and Txs require the Univalue from the RPC component. The UniValue in Bitcoin is used to encode information before communicating information with external utilities through the RPC component.

Miner <-> Txs

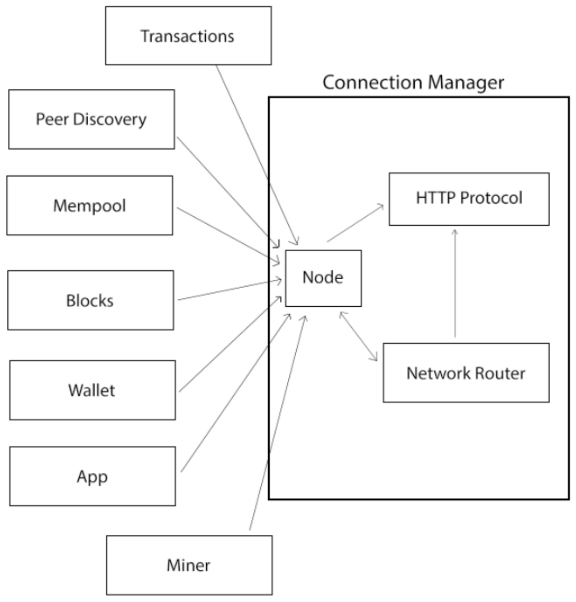
The miner needs the fee rate information contained in the Txs component, which specifies the amount of currency given to a miner for the solution to the block and also the fee cut that they get from transactions that are included within the block they mined. The Txs needs hashing information from the Miner to hash transactions and the SHA256 Uint256 to read the mining output.

App -> Txs

The app component requires information about the transaction fee rate to estimate the smart fee and as well show the fee to the user through the interface. feeRate types CFeeRate at sendcoinsdialog.cpp, and estimateSmartFee types CFeeRate at interfaces.cpp.

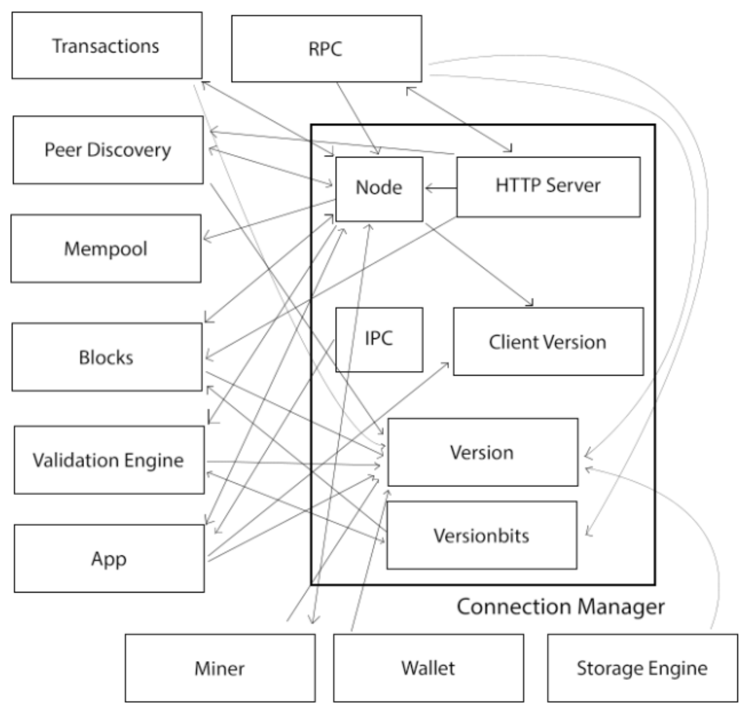
Subsystem Analysis of Connection Manager

Figure: Conceptual Architecture



We derived the conceptual architecture by utilising Cypherpunk's documentation on Bitcoin. Since every system on the Bitcoin network is a node, we had a node subcomponent in the connection manager to deal with all node related functionality. Every node on the network is also a network router, so we added a network router subcomponent (Cypherpunk). Since both subcomponents so far both rely on networking functionality, this probably implies that there must be an API/Library for the HTTP protocol. Throughout the source we used, many other components relied on using functionality related to nodes. The block subsystem uses the other node's in the network to update the blockchain, so it must rely on nodes (Cypherpunk). The app component starts by finding nodes, so it must use the connection manager with node to find peers (Cypherpunk). This is the same with peer discovery. Since each node has its own mempool, the mempool component also requires the node subcomponent. The wallet component allows coin transfer between nodes on the network, so it must use the node subcomponent to interact with peers. Lastly, the miner probably depends on the connection manager to actually get a new problem to solve, so it should depend on the node subcomponent to fetch updates to the blockchain or new problems.

Figure: Concrete Architecture



Reflexion Analysis:

The first major difference we noted is that our subcomponents were from our conceptual to concrete architecture. While we were able to map the node and HTTP subcomponents, we found that the network routing functionality seems to be done in the HTTP subcomponent. In our concrete architecture we also had 4 extra subcomponents: IPC, Client Version, Version, and Version Bits. Since nodes may use different versions of the Bitcoin Client, The Client Version, Version, and Version bits are vital because on the Bitcoin P2P network, the nodes have to communicate with each other, and they often send the version numbers as the first message between themselves, which allows a connection to occur (Cyberphunk). This is the reason why there are many dependencies leading to the different version subcomponents. The IPC subcomponent depends on the App component as it overrides a function that is called at the initialization of the app. In our original conceptual architecture we assumed that node would be depending on a common HTTP protocol API/Library, however in reality the HTTP server component depends on node, as it does the network routing and requires the node interface. Node depends on client version, as it needs the Bitcoin client version to be able to function. The versionbits subcomponent depends on validation engine to check for the deployment state, with the two states being either ALWAYS_ACTIVE or NEVER_ACTIVE. The versionbits sub component depends on the block sub component. Versionbits use classes of blocks which hold information which is needed to do calculations for specific blocks.

We found in our concrete architecture that the node subcomponent actually doesn't directly communicate with the wallet. We assumed in our conceptual architecture that there would be a dependency because we thought that the wallet needed to communicate with other nodes, but this is not the case. In our conceptual architecture, we didn't include the RPC component as we didn't believe there would be a connection, but in researching for the concrete architecture we did find a dependance, as RPC depends on the Version(bits), it depends on node for the NodeContext class, and it depends httpserver as RPC uses multiple httpserver functions in the networking sections. This makes sense as httpserver serves as a common gateway for http requests for the entire project. Storage Engine was also added to the concrete architecture as a component that has a dependency on the connection manager since it depends on the version subcomponent. Validation engine depends on versionbits to get the block state from a block index. Versionbits depends on validation engine to check for the deployment state, either always active or never active. Lastly, RPC depends on version bits to get the next and current ThreshholdState

Another difference between our conceptual and concrete architectures was that the httpserver also depends on the blocks and peer discovery components. The httpserver has a dependency to blocks, utilising the BaseParams class to get the http port in binding to a specific address. Httpserver also depends on peer discovery as it needs to check if a network address is allowed on the http server for security reasons.

Finally, the biggest difference we noted was that node depended on many outside components that were already dependent on node. We found that node depends on Peer Discovery such as in the case when the program notices that a node is full. When this occurs, the node will attempt to evict a connection to make room. To evict a connection, it depends on certain Peer Discovery networking tools to hold peer nodes that are possible candidates to evict. The eviction function does this to prevent any security attacks. Furthermore, the node subcomponent contains functions that deal with the blockchain, which depend on blocks as it uses classes from blocks to hold block information. Also node depends on miner, as the node subcomponent has a file called miner.cpp, which interacts with the miner component to allow for mining on the network. In the node subcomponent, the chainstate method is used to load the block chain state. To do this, it depends on files from the transaction component. The rationale behind this dependency is that since the transaction module implements many common functions/classes used for blockchain functionality, thus it would be more efficient and prevents redundancy by using common blockchain functionality from other components. The node subcomponent depends on classes to load Wallets indirectly from functions that are implemented in the App component. Node also depends on mempool, which makes sense since each node can have its own mempool, so the node component should be able to access the current node's mempool. The final dependency for node is the Validation engine. Node depends on the validation engine as it uses one of its data classes in implementing a cache. The rationale behind the dependency is to prevent redundancy in repeating code. In our conceptual architecture we assumed that node didn't depend on any other components since when we did our research we assumed that node played a central role in the app with other components depending on it.

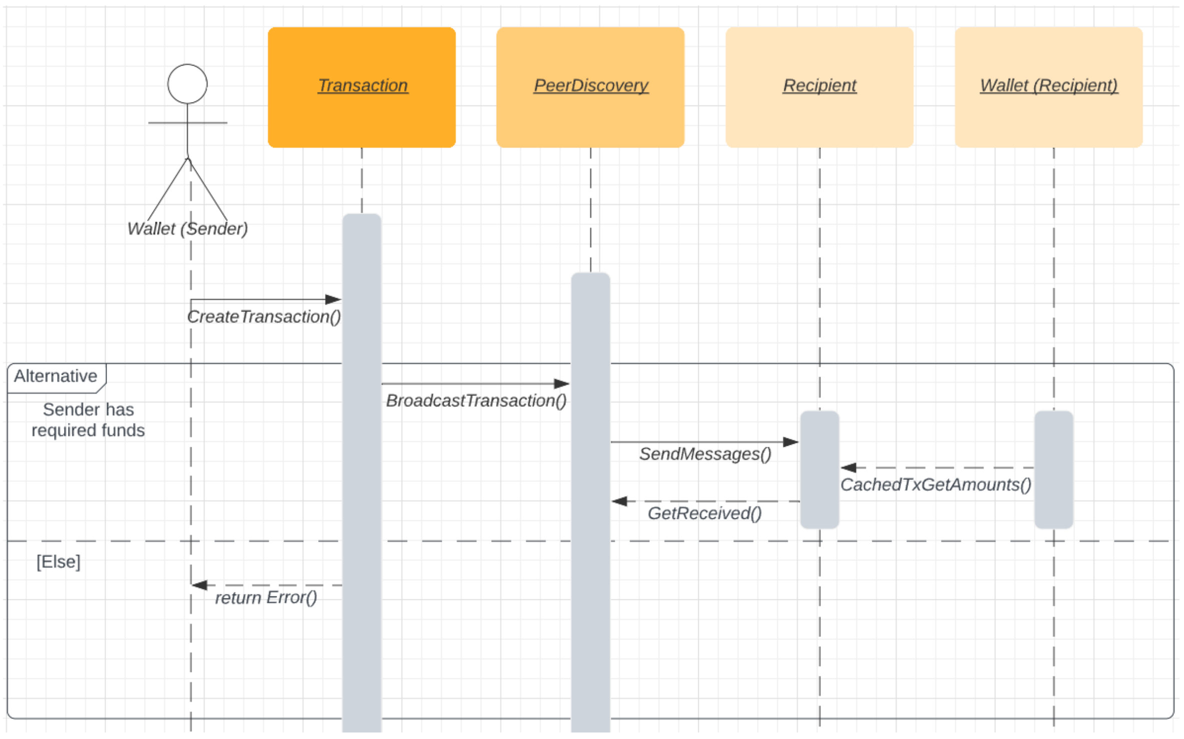
Architectural Styles:

The code base itself seems to be mainly object-oriented, since the source code is primarily written in C++, an object-oriented language. Each component is represented as a class or many classes with the dependencies between two components done through methods.

We noticed that in the source code for the connection manager, other components that depended on the connection manager were utilising many classes simply for data storage, i.e. classes with no methods. Such an example is the NodeContext class that many components depended on. This allowed other components to use networking functionality with Object Oriented abstraction.

Even though the source code has been created and updated via an object-oriented style, the connection manager itself is a part of the Bitcoin Network which connects peers via a peer-to-peer style. This is expressed many times throughout the source code. For example, if a node has maxed the amount of peers it is connected to, it kicks a peer out to make room for another.

Use Cases:



Use Case 1: Bob wants to buy a fire NFT from Alice. To facilitate the transaction, he sends her a sum of Bitcoin in exchange for the NFT. Alice accepts the transaction and the Bitcoin is added to her wallet as a UTXO.

The transaction process begins by creating a transaction in Bob's wallet. To create the transaction, the wallet first verifies that it contains enough total funds to cover the transaction amount plus any fees. If there is an error anywhere in this process, an error will be returned and the transaction will not be allowed to continue. Otherwise, the transaction is broadcast and subsequently sent along the peer-to-peer network. Then, instead of Alice's wallet being delivered a message directly, it will "receive" the funds once it checks its received transactions via CachedTxGetAmounts() and GetReceived().

Data Dictionary:

Block - A block is a collection of bitcoin transactions that have been validated and recorded on the blockchain. Each block contains the transactions, a header with other metadata, and a reference to the previous block in the chain.
Block Header - Contains information about a given block and its transactions. Block Headers are hashed repeatedly during proof of work.
Blockchain - the decentralized, immutable, public ledger of all bitcoin transactions, stored in cryptographically linked blocks of transactions.
Decentralized applications - application that runs on a decentralized network instead of a single server
Genesis Block - The first block in the Bitcoin blockchain.
Merkle Tree - also known as binary hash trees, a Merkle Tree is a tree that serves as a summary of all the transactions in a block by providing hashes for different blocks of data at its leaf nodes.
Minting - Minting is the process of creating more Bitcoin during the addition of each new block.
Mining - Validation of pending transactions within the Bitcoin network.
Miner Pools - A network of distributed miners who split rewards for a successful block solution based on computational resources provided.
Nonce - a value that is set so the hash of a block will have a number of leading zeros
Proof of Work - A form of cryptographic proof used to show that a given amount of computational effort has been used to reach a solution. The proof is easily verifiable by independent parties once it is presented. In the case of Bitcoin, Proof of Work means that a Miner has presented valid parameters to SHA256 that solve the current Block.
Full Node - A program that fully validates transactions and blocks. Almost all full nodes also support the network by accepting transactions and blocks from other full nodes, validating those transactions and blocks, and then relaying them to further full nodes.

DNS seed - A DNS server which returns IP addresses of full nodes on the Bitcoin network to assist in peer discovery.
SHA256 - The hash function used in Bitcoin Core's mining process. SHA256 always produces a 256-bit output regardless of input size. This can be thought of as a way to create unique fingerprints for every input.
UniValue: Abstract Data Type which can represent a null, string, bool, int, array container or key/value dictionary which uses JSON encoding and decoding

Naming Conventions:

UTXO - unspent transaction output
STXO - Spent transaction output
TXs - Transactions
P2P - Peer-to-peer

Lessons Learned:

Adding one additional step to our concrete architecture recovery process would have gone a long way in having a more comprehensive analysis. We derived our concrete architecture by first sorting the majority of the files into our conceptual components based on name/code analysis and interdependencies. After that we created a matrix to see which components had dependencies with which other components. We eliminated the dependencies that were clearly due to a missorting of files by relocating them and then immediately started writing justifications for the connections with the most number of dependencies. If we were to do a second pass where we think about any additional components we might want to add to our conceptual architecture we may have been able to eliminate more dependencies. By sticking rigidly to our initial concrete architecture we may have created more work than necessary.

The bitcoin core source code is not well organized. A lot of files are just placed in the src and not in a sub folder. This made it more difficult to create a good concrete architecture since we had to go from file to file to sort them and could not sort based on folder. This lack in organization is most likely caused by the fact that the system is open source and has over 900 contributors which can make it hard to be kept well organized. In the future, if we work on a big system, keeping it organized will help create the concrete architecture and make it easier for other developers to comprehend the code better and faster.

Conclusions:

A summary of your key findings and proposals for future directions.

Using the SciTools software Understand, we were able to recover the concrete architecture of Bitcoin Core and perform a reflexion analysis on discrepancies observed with our conceptual architecture. We were able to identify many divergences and provided justification for their presence when possible.

Some files are impossible to sort into any sub component as they are used frequently across the software such as files related to hashing and sha256. To surmount this, we could have added a "Utilities" section to our conceptual architecture, and designed our conceptual architecture such that all other subsystems depended on this section.

Other files were easy to sort but still resulted in dependencies not present in our initial proposed conceptual architecture, This can be seen clearly in the case of Blocks and Txs, where every single other component has at least one dependency on these two.

After doing a reflexion analysis on the concrete and conceptual architectures for the connection manager, we found there were many differences both in the entities and dependencies, as additional features were present in the actual source code compared to our research conceptual architecture. These additional features we found were useful in the actual source code and showed how the conceptual architecture does not always work out to be the end product.

Through the derivation process, we learned many lessons about the difficulties of matching our conceptual architecture to the concrete architecture, and realised how the oversimplified conceptual view does not encapsulate the complexities of a large open-source system like Bitcoin Core. Based on the lessons learned from this analysis, we suggest that future efforts could involve a second pass to refine the conceptual architecture by considering any additional components that may help reduce the number of dependencies. This approach could lead to a more streamlined and accurate representation of the actual implementation.

References:

Bitcoin Core architecture. (n.d.). Bitcoin Core Architecture. https://jameso.be/dev++2018/#1

Index. (n.d.). Bitcoin Core. https://bitcoincore.org/en/doc/24.0.0/

Bitcoin Core: Main Page. (n.d.). Bitcoin Core: Main Page. https://doxygen.bitcoincore.org/index.html

B. (2023, March 24). GitHub - bitcoin/bitcoin: Bitcoin Core integration/staging tree. GitHub. https://github.com/bitcoin/bitcoin

Chapter 3: 'bitcoin core: The reference implementation'. Chapter 3: 'Bitcoin Core: The Reference Implementation' · GitBook. (n.d.). Retrieved March 24, 2023, from https://cypherpunks-core.github.io/bitcoinbook/ch03.html