CISC 322 - A2 Report: Concrete Architecture of Bitcoin Core

Group 27 - Based

March 2023

Amy Cui (19ayc1@queensu.ca)
David Courtis (20dhc@queensu.ca)
Jagrit Rai (19jr28@queensu.ca)
John Alajaji (18ja19@queensu.ca)
Logan Cantin (logan.cantin@queensu.ca)
Matthew Vandergrift (19mwv1@queensu.ca)

Abstract

In this report we will be delving into the concrete architecture of Bitcoin Core. This will be done by first updating our conceptual architecture based on our feedback and better understanding of the Bitcoin Core system, which we will then use alongside the *Understand* software tool to map the files in the Bitcoin Core source code to the components in our updated conceptual architecture. From this we will be able to extract the concrete architecture of the system. We will then compare our obtained concrete architecture with our updated conceptual, and investigate where these two diverge. For each such divergence the reasoning behind it will be studied, based on comments in source code, documentation, and git commit information. These divergences then informed changes to the conceptual architecture, which are described in depth throughout this report.

We will similarly study the peer to peer network interface component of the system (P2P Network Interface), by performing reflexion analysis. Since no conceptual architecture for the peer to peer interface was proposed in assignment 1, one is proposed in this report. A concrete architecture is then presented contrasted with our conceptual using reflexion analysis. Additionally, we will investigate the use cases of discovering new peers, as well as receiving and rebroadcasting blocks, which are both relevant to this subsystem in particular. Finally lessons learned are presented as a guideline for how other students or our own group could better conduct this process in the future.

1 Introduction

In most software projects the resulting product is often in some way different from its initial conception. There are many reasons for this; for one the developers may realise along the way of a better way to construct their product, but also many changes may simply be due to time constraints. The latter can result in what is called "technical debt" in which the developers implement a given feature quickly, at the cost of worse project organization and likely more time debugging down the road. Hence, it is certainly not surprising that the concrete architecture for a project may be very different from its conceptual architecture; which is what we will be studying in this paper, specifically in the context of Bitcoin Core.

In the upcoming sections we will be revisiting our conceptual architecture (from A1), and updating it with components and dependencies which we missed or which need to be modified. Using our updated conceptual architecture we will then utilise the *Understand* software to map the files of the Bitcoin source code to our conceptual components, in order to extract the concrete architecture of the system. As this process can in fact be rather difficult (especially with a relatively large system such as Bitcoin Core), we will also be explaining our derivation process for obtaining the concrete architecture, as well reevaluating our process at the end of this paper, to better understand how we could have proceeded in a more effective manner. Once the concrete architecture of the system has been obtained, we will then be performing a reflexion analysis to better understand how it differs from our conceptual architecture, and why.

Finally, we will finish our study by taking a deeper look at the peer to peer interface component of our system (P2P Network Interface). To do this, we will first construct its conceptual architecture based off documentation and

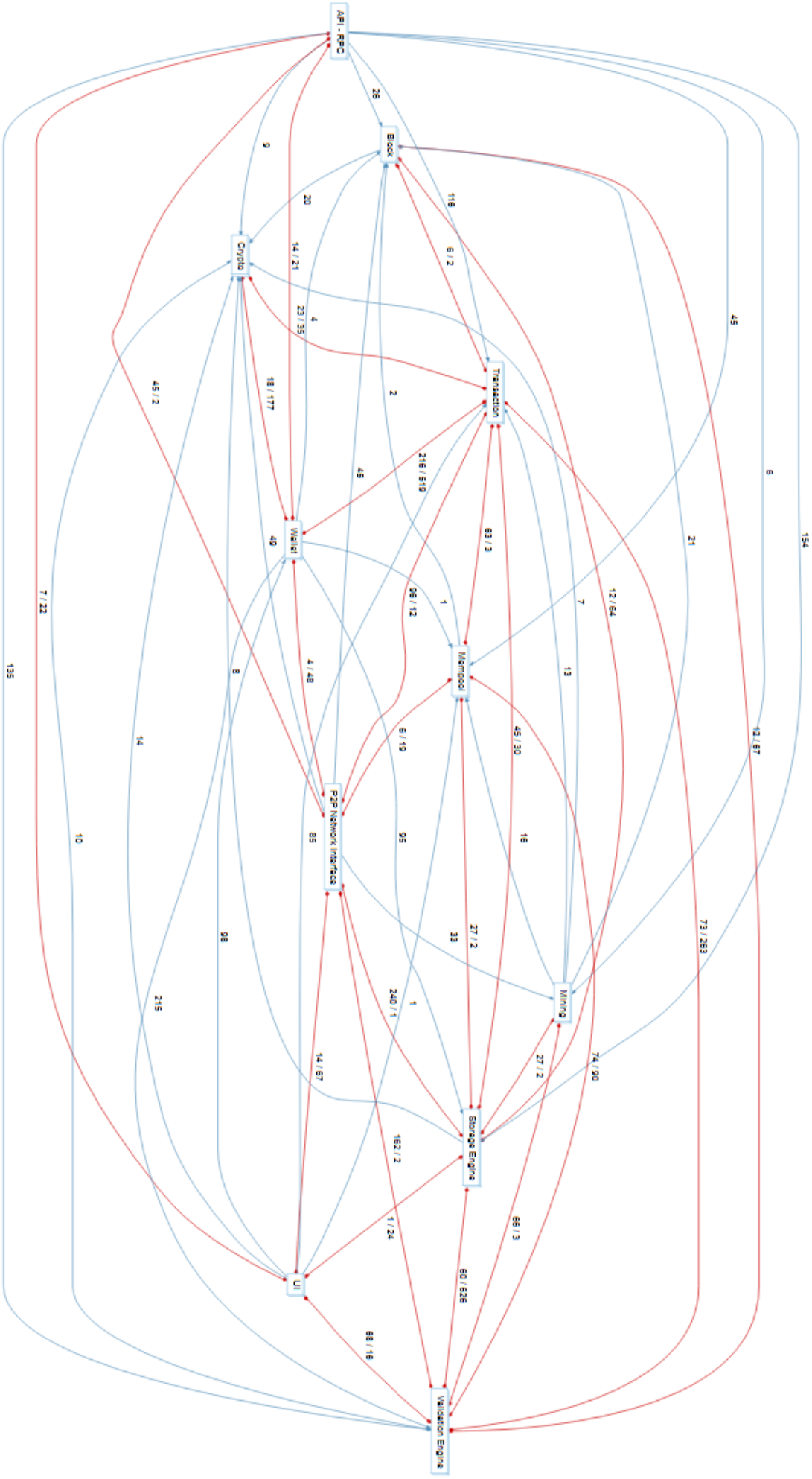

Figure 1: Understand Graph without util and init

pertinent information originating from various sources, and then perform a reflexion analysis to extract its concrete architecture and once again explore the unexpected dependencies obtained (once again this will be done with the help of *Understand*). We will then finish this section off by considering two use cases relevant to this subsystem, and constructing the corresponding sequence diagrams. The use cases we will be considering are discovering new peers, as well as receiving and rebroadcasting blocks, which are both particularly relevant to this subsystem.

Let us now begin by taking a look at our updated conceptual architecture for Bitcoin Core.

2 Updated Conceptual Architecture

The conceptual architecture was updated according to the principles of software reflexion as outlined in Software Reflexion Models: Bridging the Gap between Source and High-Level Models by Murphy [2]. This means that a cyclic process of mapping source code to conceptual architecture then updating the conceptual architecture was followed. The following is a boxes and arrows graph of the conceptual architecture generated through our reassessment of our conceptual connections based on a deeper analysis of the bitcoin core. Connections that were modified since our last report will be explained in detail.

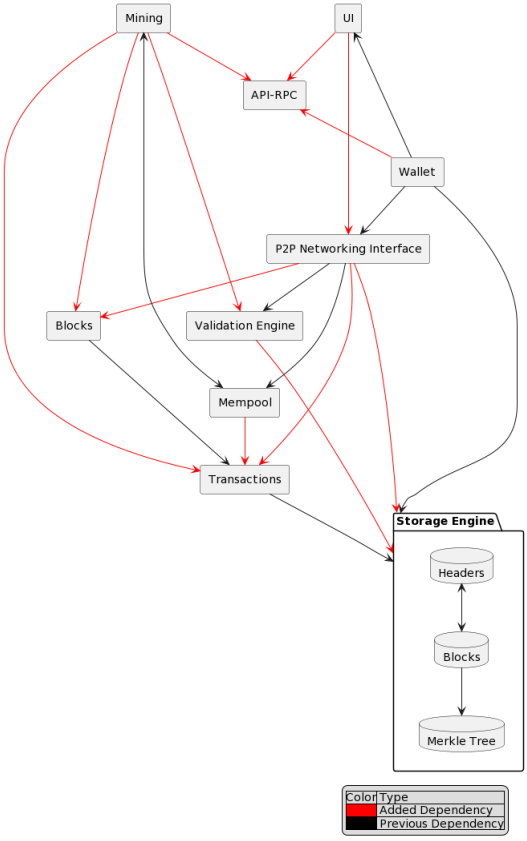

Figure 2: Updated Conceptual Architecture

There were two different types of changes which were made to the conceptual architecture: our modified components and modified/additional dependencies. These are expanded upon below.

2.1 Modified Components

The updated conceptual architecture, as displayed in Figure 2, is a result of our thorough analysis of the dependencies within the Bitcoin Core system. It is also based on the feedback we received for our previous assignment, and our increased understanding of the system. We will now go through the specific changes we made to the components of our conceptual architecture.

2.1.1 API-RPC

API-RPC is the Application Programming Interface-Remote Procedure Call component. This component was not included in our previous conceptual architecture, however is certainly an important part of the system as it allows developers to send requests to the Bitcoin software, as well as provides certain functionalities to nodes (e.g., used by miners when constructing a new block).

2.1.2 Storage Engine

The storage engine is a new component that was not included in our previous conceptual architecture. This component is responsible for managing the storage and retrieval of all data related to the bitcoin network, such as transactions, blocks, and other network data (e.g. list of known peers). The storage engine component provides functionality for nodes to retrieve and send data to other nodes in the network, by allowing nodes to query the database for the required data. The storage engine is also responsible for maintaining the network's state, which includes information about the current state of each transaction, the unspent transaction outputs (UTXOs), and the status of network nodes.

2.1.3 Wallet

In our previous conceptual architecture, our Wallet component was split into two main pieces the wallet file and the wallet program. In the updated conceptual architecture it is one component. The main reason for this change is since previously the wallet file was responsible for the storage of keys generated by the wallet program, however this responsibility is performed by the Storage Engine (which also explains the new dependency now present between the Wallet and Storage Engine). Another reason for this change is simply that the division between the management of key storage and actual key storage is not present in the actual implementation of the system.

2.1.4 UI

The UI used to be considered just the wallet UI, however upon seeing how the UI supplied messages to the user given to it by components throughout the system it became clear that the UI component exists outside of the wallet. Hence we truly made the UI a standalone component.

2.1.5 Validation Engine

This component is responsible for ensuring that the data handled by the system follows the rules and specifications necessary for the bitcoin core system, and for the overall functioning of the blockchain. The validation engine is a modified version of our prior Message Verification component. In this architecture, the validation engine has a narrower scope; it is now only responsible for ensuring the validity of a block or transaction before adding it to the blockchain or rebroadcasting it. The other responsibilities from the prior architecture, such as Broadcast storm prevention and Packet verification, are handled by sub-components within the Peer-to-Peer Networking Interface component, which we discuss later.

2.1.6 Other changes

In our updated conceptual architecture, we decided to simplify the structure by removing the contracts component. This was done as Bitcoin contracts are a specialized sub-component of transactions, and their usage is too low-level to be included in the high-level conceptual architecture. Therefore, we streamlined the architecture by focusing on the core features and functionalities that are essential to the Bitcoin network's operation.

Additionally, we removed the blockchain component, as we came to understand that although the blockchain is an essential part of how the system operates, it is not a component of the conceptual architecture, rather, it is an implicit object within the architecture (i.e., more of a database stored in the Storage Engine). In our previous architecture, it had subcomponents such as blocks and merkle trees; in our updated conceptual architecture, we moved these sub-components into the storage engine,

Additionally, the block constructor component from our previous architecture was renamed to "Miner"– however, it still maintains the same functionalities and dependencies relating to the process of minting/creating new blocks.

2.2 Modified Dependencies

Miner → Validation Engine : As discussed in Section 2.1, the scope of the Validation Engine was narrowed be responsible for ensuring the validity of a transaction and blocks. Specifically, it is responsible for ensuring the cryptographic validity of a transaction, as well as verifying that the UTXOs aren't double spent.

UI → P2P Network Interface : The UI controls the P2P networking interface (by allowing the user to change networking related settings, like maximum bandwidth allowed or to change the physical network interface that is being used).

UI → RPC-API : The UI can access the RPC-API, which is the external interface which allows other programs to interact with Bitcoin Core programmatically.

Wallet → RPC-API : The wallet also depends on the RPC-API, so that other external programs can gain access programmatically to the wallet and wallet-related functionality.

P2P Network Interface → Transaction : The full node receives transactions to be put into mempool and mined, and hence relies on the transactions component.

P2P Network Interface → Block : When a new block is mined by some other full node it is broadcasted to the blockchain, which must first pass through the P2P Network Interface. To understand what is being received, the P2P Network Interface needs to parse incoming/outgoing blocks, and hence it should not be surprising that it would rely on the blocks component.

Mining → Block : The mining component takes transactions and builds blocks, hence it relies on the block component. The reason this was not initially apparent is that we previously distinguished being block construction and blocks, however with our refined architecture the mining component simply builds block, this of course makes use the block component.

Mining → Transaction : It was previously understood that the miner generates blocks, however since blocks are composed of transactions this means that miner depends on transactions. Once it was seen that the block did not encapsulate all transaction data and operations it became clear that the miner would have to interact with transaction data. This is seen in the generation of new blocks, for instance the essential CreateNewBlock function in Mining calls CMutableTransaction which is a function provided in transaction.

Mining → API/RPC : Mining depends on API/RPC since miners use RPCs during the mining process. For instance, a miner's "mining software periodically polls bitcoind for new transactions using the "getblocktemplate" RPC, which provides the list of new transactions plus the public key to which the coinbase transaction should be sent" [1].

Mempool → Transaction : The mempool consists of transactions which will be mined and put into a new block. Since transactions make up the mempool, by definition mempool relies on transactions.

Validation Engine → Storage Engine : In order for the validation engine to verify that UTXOs are not double spent, it must check the blockchain which is stored in the Storage Engine.

P2P Network Interface → Storage Engine : Since the P2P Network Interface is now responsible for verifying that messages are safe, then we now have a new dependency between the P2P Network Interface and the Storage Engine. This is since for example when verifying messages the system will first verify that the message was not sent from a blacklisted peer, and such a list is stored in the Storage Engine.

3 Derivation Process

Now before moving on to our section on the concrete architecture, let us walk through the process we followed in deriving it. This process mainly required using the *Understand* software by Scitools, as well as it's integrated bit blame functionality, to map files from the Bitcoin Core source code to the components in our updated conceptual architecture (as well as a few more components such as an initialization component, which is only really part of the concrete architecture); however this is easier certainly easier said than done. In our Lessons Learned section we will go into more depth regarding why we found this to be so difficult, however in brief, due to the large volume of files and the oftentimes poor documentation of them (at least to a relative outsider such as ourselves), this process was definitely rather complicated.

The way we mainly proceed in mapping the files was first to map folders which seemed to obviously relate to a particular component simply from its name. For example, we quickly mapped the rpc folder in the source code to the API/RPC component of our architecture, the Init folder to our newly added Init component for our concrete architecture (see next section), and so on. We did however quickly verify that these files did indeed seem to serve the purpose that their name suggested (since it is possible that some folders could be poorly named), but in general this part of the mapping did not cause much trouble. Where things became more complicated though was for mapping all the other folders, as well as the large abundance of loose files. Our approach for these folders/files was then first to look at the project's github for possible documentation. In the case of some folders, the documentation explaining the purpose of the folder is given (e.g. for the zmq folder), however in most instances this is not the case. Our next step was then to take a look at the code itself, and look through for helpful comments. For many files this ended up solving our problem, however for many more we also required doing further research online to see if any additional information could be found about the files/folders in question.

One key question which I think is worth bringing attention to here is, if a file seems to map to multiple components in our architecture, then where should we put it? A good example of this is in the wallet folder of the source code there is a folder dedicated to RPCs used for the wallet? So where should it get mapped to? To the Wallet component, or to the RPC component? The general methodology we followed was to map the files to whichever component of the architecture they seemed most related to. So for the example just given, we mapped this rpc file to the Wallet component, since despite it being an RPC, it is only used by the Wallet. The reason we chose to proceed in this way, is so that if another component interacts with the Wallet component using something found in this wallet rpc file, then we would have a dependency between this component and Wallet, instead of between it and RPC.

Finally, once the files were all mapped using *Understand*, we were left to extract the concrete architecture from the dependency graph. As can be seen from **1** our resulting dependency graph is almost a complete graph; however many of the edges only have a few dependencies associated to them. In some instances, these edges are still important to consider, however in most cases we found them to be very superficial; typically being due to an include statement which is not really used, or in many other cases due to some test/block code which calls files in other components. Therefore, we filtered through the dependencies in our obtained graph, and tried to only consider the most important ones, which we will now discuss in our section on concrete architecture.

4 Concrete Architecture

The following is a visualization of the concrete architecture of the bitcoin core subsystem,

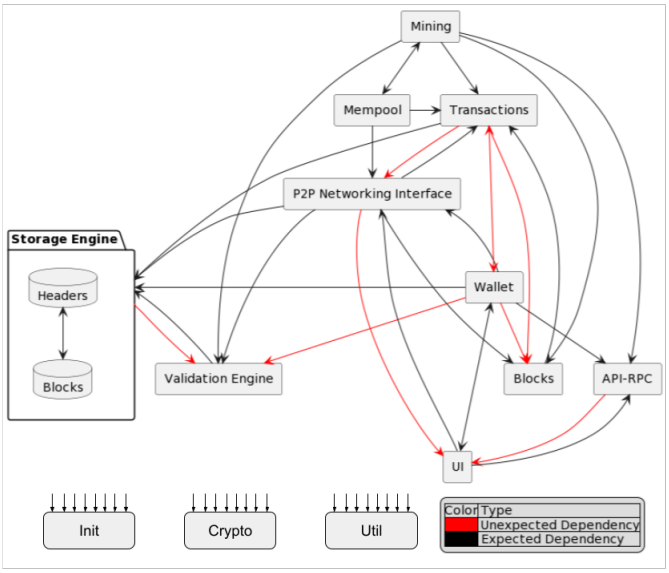

Figure 3: High-Level Concrete Architecture

Note that this architecture is in some ways quite similar to our conceptual architecture and effectively follows the same general peer-to-peer architectural style, however it does have some key differences. We found that the concrete architecture seemed to incorporate some behavior akin to a publish-subcribe architecture (pub-sub). This was especially apparent in the zmq folder which is an important part of the P2P Network Interface component (covered

in section 5), which acts asynchronously when relaying real-time events within the Bitcoin Core software or to external applications.

Some specific notation used for this diagram is the use of multiple disconnected arrows to indicate complete connection, this notation was seen in the concrete architecture of the Linux operating system [5]. Firstly, it is immediately apparent that everything depends on Init since it initializes the sub-systems. Additionally since Util contains common used types and operations all components end up depending on it in one way or another. The reason we chose to represent the component which depend on Crypto as disconnected arrows is because virtually every subsystem depends on Crypto from a concrete perspective. This allows the graph to remain uncluttered.

4.1 Concrete Specific Components

- The first concrete component we had was an initialization component. We found that files which mainly booted up a component or set it's beginning governing rules were pervasive throughout the system. Many of these files set up parameters of the system such as chainparamsbase.h. Others adapted other parts of the system to work on the platform which the system was begin launched from such as randomenv.cpp . We determined that all files which declared base line rules, or which started a sub-system into operation were to be included in this component.

- The second component we had was a util component. This is essentially a component for common operations done throughout the system by other sub components. This has no main clear purpose and hence not relevant to a conceptual architecture but it did correspond to many of the files which made up the source code. The files we decided ought to map to this component were mainly data types, logging files, and operations frequently called by every component.

- We defined block as a component outside of the storage engine for our concrete architecture. This might seem confusing at first since we have blocks as a database inside of the storage component, however from the concrete perspective there is a distinction between the two. The block component defines the block itself and the operations which can be done on the block. The blocks database in the storage engine shows that the storage engine stores the blocks which make up the block-chain on the client's machine.

- From the conceptual level transactions are a key data piece stored inside blocks, and hence in the storage engine. At the concrete level, transactions represent a collection of different data and also operations which can be done on that data. The collection of code written to operate on transactions was mapped to this component. This encapsulates how the developers use transactions in all other components hence this component should exist.

4.2 Divergent Dependencies

Transaction → Wallet : When transactions are created they use data which is stored in the wallet. Some important constants are keep in wallet related files, examples are CAmount (which determines amount of staosihis in one coin) and MAX_Money (which defines the max amount of satoshis which can be dealt with at once).

Wallet → Block Upon considering the basic operation of the bitcoin core system and from reading high level developer guides it appeared that the wallet handled key generation and management and hence no dependency with the blocks was expected. The major source of this dependency was that the wallet backed up data based on block data. For example backup.cpp which is a wallet file uses merkleblock.h

Wallet → Verification Engine This dependency comes from the wallet's use of the consensus rules to ensure that the fee rates, block rewards, max transaction size, and spending limits it is storing are inline with the rest of the system. The files which store this data rely mainly on validation.h and consensus.h

Transaction → Block One the responsibilities of the transaction component is to get a log of transactions. Since transactions reside in the blocks, the transaction component must search through the block. The reason this dependency is unexpected is that when the transaction component was conceived of, the role of getting transactions was not considered.

API-RPC → UI This dependency was deeply unexpected, since RPC-API provides the functionality for developer's to interact with and modify the system, but UI exists for ease of use of the end user. These two goals seem incongruent and hence no dependency between the two components was expected. It turns out that this dependency was mostly due to the RPC console which allows access to API-RPC functionality directly on the system. This is why the dependency is seen in the concrete architecture.

Wallet → Transaction : A large amount of the direct reliance on transaction from wallet comes from the storage and manipulation of partially signed transactions. The wallet handles these partially signed transactions very frequently making use of many functions suchs as PSBTInputSigned, PSBTinput, PartiallySignedTransaction,SignPSBTInput, FinalizeAndExtractPSBT etc. This was unexpected because from a purely conceptual perspective there is no distinction between partially signed transactions and all other transactions. In the actual implementation there is and so the wallet must really on this part of the transaction component.

Transaction → P2P : This is because transaction makes use of information about the nodes from which it came. To get this information it uses NodeID which is a feature provided by the peer to peer interface.

P2P → UI : When establishing or managing connections issues such as interfering connections will occur, these issues need to be communicated to the user of a full node. To notify users the peer to peer interface sends a message to the user interface. This use of the UI to communicate information is done mainly through the function MSG_ERROR by net.cpp. This is unexpected since the use of case of displaying network challenges was not considered until the source code was viewed.

Storage Engine → Validation Engine : The storage engine is responsible for managing the data storage and retrieval operations for the blockchain, or the distributed ledger that records all Bitcoin transactions. The storage engine parses the data from the blockchain and as such needs to index and locate blocks. To ensure it is distinguishing between blocks according to the current rules of the system it uses the validation engine. This can be seen in files like base.h which index blocks and all make use of the validation interface to handle block objects. This dependency is novel because validation and indexing appear conceptually to be unrelated however in practice they are related.

5 Peer to Peer Network Interface Component Reflexion Analysis

The second level subsystem which we will be considering in this study is the peer to peer network interface component (P2P Network Interface), which can be found as part of our architecture for Bitcoin Core. The system's main functionality is communicating with the peer to peer network, which includes (but is not limited to) tasks such as peer discovery and maintenance, as well as sending transactions or blocks to other peers in the network.

5.1 Conceptual Architecture

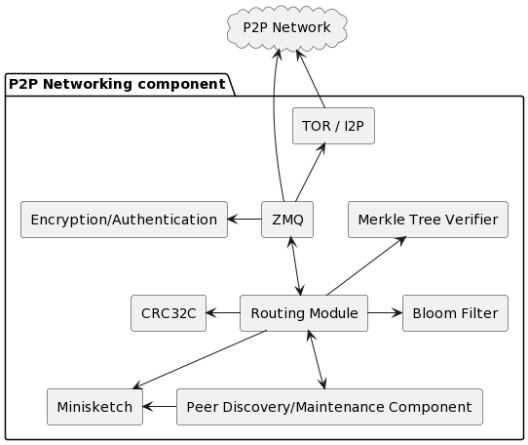

Figure 4: Conceptual architecture of the P2P Network Interface component

Using **Figure 4** as a reference, let us delve deeper into the tasks performed by each sub-component, as well as their connections and dependencies.

5.1.1 Sub-Components

- **CRC32C**: This is an external library that is responsible for doing error detection in data transmission and storage. It is a checksum algorithm that utilizes the 32-bit Castagnoli cyclic redundancy check.

- **Routing module**: This module is the brains of the peer to peer networking component. It is responsible for routing all the packages that are received and transmitted. It is also responsible for deciding if and when to rebroadcast messages. This module is critical to the operation of the network: if the rebroadcasting logic is faulty, then there can be broadcasting storms that destroy all the bandwidth, or messages might not get propagated across the network properly.

- **Peer discovery and maintenance**: This module is responsible for initiating, reciprocating, and maintaining peer connections to the network. It is also responsible for managing malicious peers, avoiding denial of service attacks and punishing them by ignoring their messages.

- **TOR and I2P**: Allows the user to connect to the network in a more anonymous way. Specifically, makes it harder for a man in the middle (MITM) attack to occur by disassociating plaintexts from the nodes that send them. This is useful specifically for simple payment verification (SPV) nodes, who, in simple terms, only send and receive transactions that pertain to them. A MITM attack could be used to de-anonymize the identity of an SPV node, reducing the anonymity of the network. This module is not commonly used if the node runs on the surface web, or the clearnet, therefore, it is excluded from many use cases.

- **Bloom filter**: Provides privacy for SPV nodes by allowing them to partially obscure the transactions that they are interested in from other peers. This also helps with preventing de-anonymizing an SPV node by telling the nodes exactly which public keys it owns. The bloom filter provides privacy by more loosely specifying the public keys of interest for an SPV node, instead of supplying the precise keys it owns to its peers. This mechanism provides a tradeoff between privacy and bandwidth saving (looser filter = more privacy but more bandwidth, and vice versa).

- **Authentication and Encryption**: This is a bitcoin enhancement (BIP-150/151) that allows connections between peers to be encrypted and authenticated. Again, this feature is useful mostly for protecting the privacy of SPV nodes. By default, messages between Bitcoin nodes are unencrypted and unauthenticated. Usually, this doesn't cause an issue: full nodes listen for all messages, from all sources. In the case of SPV nodes, only specific messages are requested, so if an eavesdropper or a MITM attacker gets access to the messages that are being sent and received, it can de-anonymize the owner of the SPV node.

- **ZMQ (zero message queue)**: This is an asynchronous messaging library that forms the backbone of the P2P messaging capability. This library specializes in messaging without a message broker (a centralized server which routes messages to the appropriate sources), which is critical for relaying real-time events within the Bitcoin Core software or to external applications.

- **Minisketch**: This is a set reconciliation library, used by the P2P interface to efficiently determine differences between sets. For example, if node A and node B need to determine the difference between their set of transactions in the mempool, UTXOs, peers, etc, the naive way would be to send the whole set of data. This requires a lot of bandwidth. Minisketch allows the nodes to calculate a kind of hash (called a "sketch") that represents their set. The hash can get sent and compared. It is possible to discover differences between the sets using just the sketches, significantly reducing the cost of discovering the difference between sets.

- **Merkle tree verifier**: This is used by P2P Network Interface and SPV nodes to "verify" (in quotes because it is resilient against double spending) transactions using a fraction of the bandwidth that would be required to download the whole block. Instead of needing to send all the transaction in a block, say N transactions, all that is required is $\log_2 N$ transactions to be sent in order for it to be verified that a transaction is in a block.

5.1.2 Dependencies

Since the Routing Module is the central part of the subsystem, it should have the most connections/dependencies. Firstly, it is connected to the ZMQ component, for support (de)serializing messages and handling message receipt and sending. Additionally, it uses the Merkle Tree verifier to do simple verification of transactions, the Bloom Filter to allow for more anonymous queries, as well as the Peer Discovery/Maintenance component, which is responsible for creating and maintaining peer connections. Finally, the Routing Module also depends on CRC32C which is responsible for error connection which can occur when transmitting information over the network.

We proposed that the ZMQ can connect directly to the P2P network for sending and receiving messages, as well as having support for Encryption and Authentication, which is useful for protecting users' privacy when operating a SPV node (by default, Bitcoin messages are sent in plaintext). Secondly, it supports Tor/I2P, which allows the user to obfuscate their identity.

Finally, the last dependency which has not yet been mentioned is between the Peer Discovery Component and Minisketch. Minisketch allows users to efficiently compute the difference in various collections of data they possess, and so is very useful for peer maintenance (e.g. can be used when updating a nodes list of peers, by obtaining the list of peers of another node, and adding the difference in peers between the two lists, as new discovered peers).

5.2 Concrete Architecture

The following is a visualization of the concrete architecture of the Peer to Peer Network Interface subsystem.

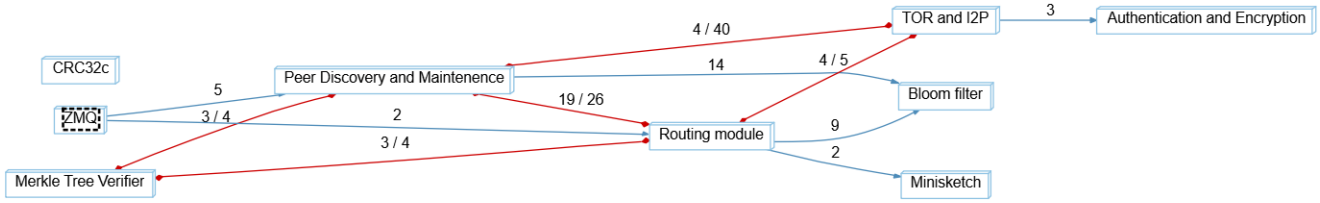

Figure 5: Understand the concrete architecture of the P2P Network Interface component, with red lines representing two-way dependencies, and arrows representing one-way dependencies between components. The numbers above these arrows represent the number of calls between the components.

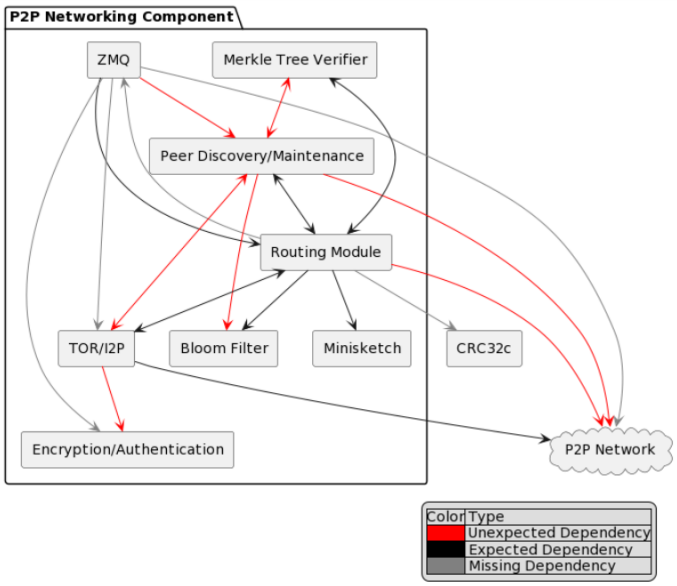

Figure 6: Overview of the concrete architecture of the P2P Network Interface component. Although the P2P network is not a concrete component, it was included to demonstrate the interaction with the network through calls in the concrete architecture.

While our conceptual components were able to cleanly represent the Peer to Peer Network Interface Component within our concrete mapping, there were many vacant or unexpected connections within our dependency graph as represented between Figure 6 and Figure 4. Many of these unexpected dependencies do not greatly impact our conceptual overview and are not difficult to integrate into our understanding as they maintain the functionality of each component; however, the ZMQ component was greatly misunderstood within our conceptual overview. Contrary to initial assumptions, this component does not deal with P2P communication or the underlying network protocols, and only deals with relaying real-time events within the Bitcoin Core software or to external applications on the host.

What is notable is the increased interconnections of the Peer Discovery and Maintenence component due to the presence of net* files, responsible for networking with the network directly, being distributed between these components. Because of the wide scope of these files, many connections are required to maintain functionality. There are also notably

reduced interconnections within our ZMQ component due to the decreased scope of the component, as it does not handle P2P communication or the underlying network protocols.

More detail about each divergence will be explored below.

5.2.1 Divergent Dependencies

- **ZMQ → Peer Discovery and Maintenance** : The call from ZMQ to Peer Discovery and Maintenance arrises due to the zmqpublishnotifier.cpp file, which is used to specify the IPv6 address format in the error message when there is a problem with the ZMQ socket binding. This is directly related to netaddress.cpp in Peer Discovery and Maintenance because this file implements the handling of network addresses, including parsing, serialization, and validation of IP addresses and associated data. Although this is a minor dependancy, it is notably important for error handling.

- **TOR and I2P ↔ Peer Discovery and Maintenance** : The TOR / IP component, which includes the files i2p.cpp and torcontrol.cpp, plays an important role in enhancing the privacy and security of the Bitcoin network. These files happen to be directly linked with Peer Discovery and Maintenance component, specifically the net*.cpp files, as these files are responsible for handling various aspects of the peer-to-peer network communication, such as connecting to peers, sending and receiving messages, and managing connections. As a result, they are closely linked to the Tor and I2P files, as these files help provide secure and anonymous communication channels between peers.

- **TOR and I2P → Authentication and Encryption** : The torcontrol.cpp file includes the crypto/sha256.h header, which provides the necessary SHA256 hashing functions. The hashing function is used in the TorController::AuthDone() method to generate the onion address of the hidden service.

- **Peer Discovery and Maintenance → Bloom Filter** : The peer discovery component, banman.cpp is responsible for managing the banning of misbehaving peers, while the net files are responsible for handling various aspects of the peer-to-peer network communication. The bloom.cpp file, on the other hand, implements the Bloom filter functionality, which is used by Simplified Payment Verification (SPV) clients to request only relevant transactions without revealing their full set of addresses. The connection between banman.cpp, net files, and bloom.cpp is mainly due to the fact that Bloom filter-related messages are part of the P2P communication and protocol. When a peer receives a Bloom filter-related message, such as a filter update or a request for filtered blocks, it needs to process and respond to the message accordingly. This involves interactions with the peer discovery and maintenance components to ensure that the communication is managed correctly, and with the ban manager to penalize any misbehaving nodes.

- **Peer Discovery and Maintenance ↔ Merkle Tree Verifier** : The main points of connection here are between net_processing and net files in Peer Discovery and Maintenance, and the headerssync.cpp and headerssync.h files in Merkle Tree Verifier due to the exchange of block headers between peers and the processing of those headers. When a new node joins the Bitcoin network, it needs to synchronize with the current state of the blockchain by obtaining the necessary block headers and block data. The headerssync.cpp and headerssync.h files are part of the Merkle Tree Verifier component, which is responsible for verifying the integrity of block headers and transactions using Merkle trees. As the node receives block headers from its peers, the Merkle Tree Verifier checks the validity of these headers, ensuring they follow the consensus rules and form a valid chain. This process is crucial to prevent nodes from being misled by malicious peers that may provide invalid block headers.

- **Routing Module ↛ CRC32C** : CRC32c is actually not directly used within the core Bitcoin protocol itself, instead it is largely relevant in the LevelDB database that Bitcoin Core uses for storing the blockchain data, such as the UTXO set or transaction indices. Instead of having this component solely involved in error detection in packets that are involved in the peer to peer networking, the error detection is infact spread out over multiple net*.cpp files within the Peer Discovery and Routing Module components, which verify that the checksum for a given payload is valid using the double-SHA256 hash method.

- **Routing Module ↛ ZMQ** : The ZMQ is not the main gateway between the p2p network and the routing module. This role is adopted by various files in the Peer Discovery/Maintenence and Routing module components. The primary files that instigate this connection are the net*.cpp and net*.h files, which deal with the core networking functionality, including establishing connections, sending and receiving messages, and managing connected peers, as well as providing low-level networking primitives, such as creating sockets, resolving hostnames, and managing network timeouts. The ZMQ instead, will act as a message queue and notifier for relaying

real-time events, such as new transactions or blocks, to external applications that may be monitoring the Bitcoin network or performing additional processing. Although this sounds similar to the routing component, ZMQ does not regard the transmission of bulk data, instead, it represents only the messaging infrastructure for relaying real-time events within the Bitcoin Core software or to external applications that interact with the node.

- **ZMQ ↛ TOR and I2P / P2P Network** : The TOR/I2P component provides anonymous communication capabilities for nodes in the Bitcoin network, whereas ZMQ component provides an efficient and asynchronous messaging infrastructure for relaying real-time events within the Bitcoin Core software or to external applications. ZMQ does not deal with P2P communication or the underlying network protocols. In the case of Bitcoin Core it is a socket which other applications can connect to and get notifications of new events in real time, without having to repeatedly ask the daemon if there are any new events.
- **ZMQ ↛ Encryption/Authentication** : While BIP-150 (peer authentication) and BIP-151 (peer-to-peer encryption) have been proposed to enhance the security of connections between Bitcoin nodes, they have not been fully integrated into the Bitcoin Core implementation as of yet. This indicates that this connection may arise in the near future,
- **Peer Discovery and Maintenance / Routing ↛ P2P Network** : Interface with the P2P network occurs not only within the tor and I2P components, but are also present in the general networking functionality seen within net* files. Both sets of files interact directly with the Bitcoin network by managing connections, exchanging messages, and maintaining the network structure. While the net* files handle the general P2P communication layer, the i2p and torcontrol files specifically deal with integrating I2P and Tor for enhanced privacy and anonymity.

5.3 Use Cases

5.3.1 Discovering New Peers

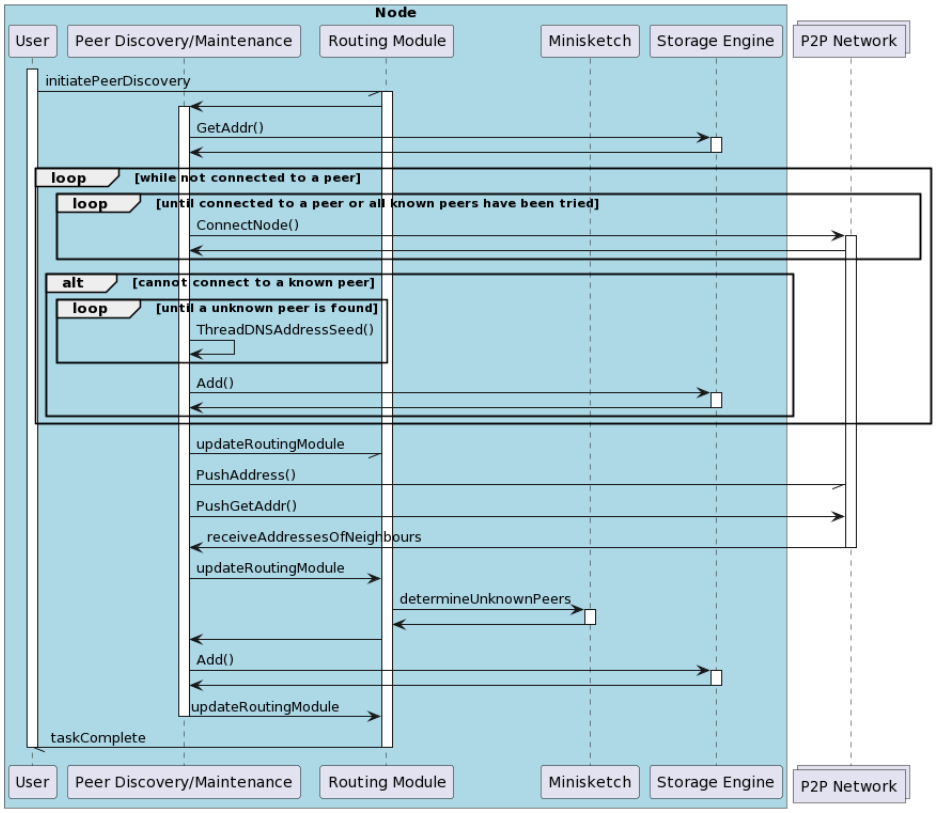

Figure 7: Sequence diagram for peer discovery

In this section we will be discussing the use case of discovering new peers in the bitcoin network. Note that for this use case, we will specifically be considering the situation in which the node initiating the peer discovery already knows at

least one other peer, and additionally that the node is running on clearnet. This second assumption essentially means that the node is not utilizing the I2P or TOR components when connecting to the network.

To begin peer discovery, the node must first connect with at least one other peer. This is done by first retrieving from the storage engine the list of known peers (via the GetAddr()), and then attempting to connect to each of them. If this is unsuccessful, then the node will need to then query DNS seeds (using ThreadDNSAddressSeed()) to find a new peer to connect to. Once this is done, the new peer will be added to the list of unknown peers, and the node will try to connect to it (using ConnectNode()). This process will be repeated until a connection has been established.

Once a node has connected to at least one peer, the rest of the process of peer discovery is in fact quite simple. The idea is that the node will then transmit its IP address to all its neighbours via an addr message (using PushAddress()), along with a getaddr message to retrieve the IP addresses of all its neighbours peers (using PushGetAddr()). Once these addresses are received, the node will then use the Minisketch component to retrieve from the obtained addresses only the addresses which it does not already know, which it will then store in the Storage Engine (using Add()).

5.3.2 Unsolicited Block Push

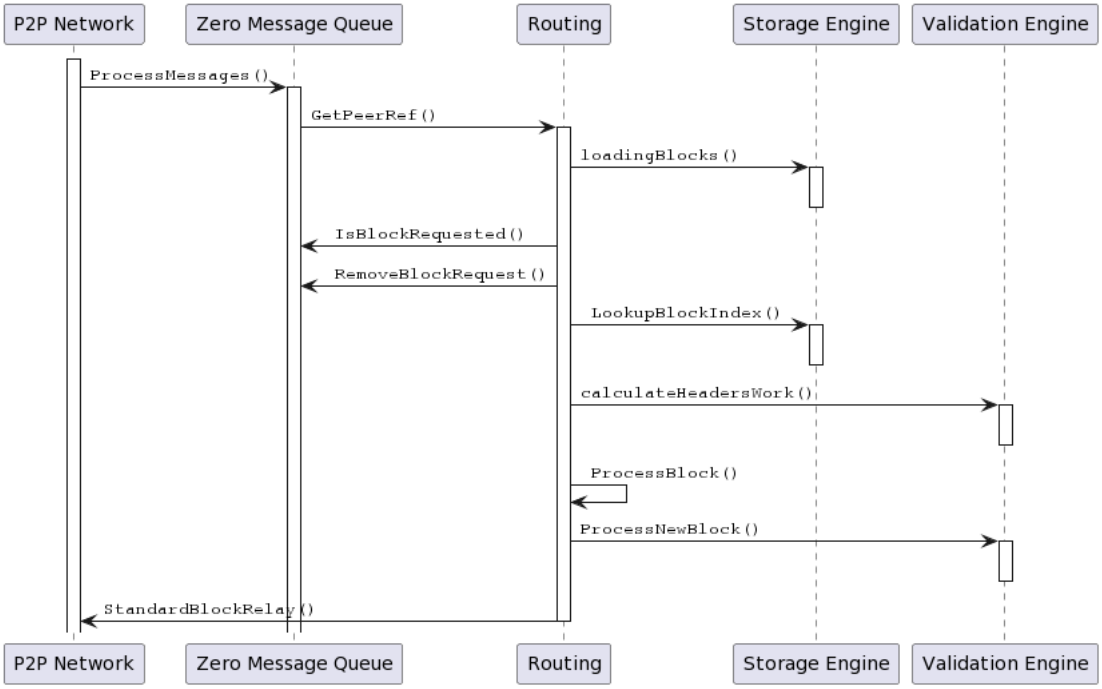

Figure 8: Sequence diagram Unsolicited Block Push

In this section, we consider the use case of full nodes receiving an unsolicited block push. Some miners will send unsolicited “block” messages broadcasting their newly-mined blocks to all of their peers. When such a message is sent, the peer to peer network interface of a Full Node peer stores processes the message, storing a reference to the peer who sent it. Next, the Storage Engine component is activated twice. First, it checks if the node is currently loading blocks, in which case the block should be ignored. Second, it looks up the previous block in the chain using the hash included in the received block.

The Validation Engine component is then activated to check the proof of work done on block against the system’s anti-dos thresholds. The Routing component then processes the block, updating the blockchain with the most recent time a block was added, and validates it using the ”ProcessBlock()” and ”ProcessNewBlock()” functions respectively. Note that the ProcessBlock() function actually calls the ProcessNewBlock() function, ensuring that the block is validated before it is added to the blockchain. Finally, the node rebroadcasts the block to it’s peers by using the standard block relay protocol, which sends an “inv” message to each of its peers (both full node and SPV) with an inventory referring to the new block.

6 Lessons Learned

In this section we will be reflecting on what we could have done differently to further improve our project (and complete it in a more efficient manner). To be begin with, we greatly underestimated the work involved in using *Understand* to map the source code files to the components in our conceptual architecture. Perhaps this shouldn't have been so surprising, as this project focuses on reflection analysis; however regardless we likely waited a bit too long to begin this part of the project, which then ended up delaying other parts of the project (as this part of the project lies on the critical path). So in brief, we definitely should have started the file mapping for the overall architecture earlier, and additionally should have probably reached out to Professor Adams and the TAs earlier to clear up the problems we were having with this part of the project. Mapping files to components is a process which is conceptually very simple however constructing a rationale for all but the most obvious of files is a time consuming process. The names of files/folders were often abstract of misleading, and there was very little documentation. Additionally the git logs were helpful but often written in extremely technical language and not helpful for mapping files to conceptual concepts. The knowledge of how complex this process is would have helped us better plan, but also would have motivated us to better split up and divide this task to speed it up.

Another key lesson learned was the importance of rigorous documentation at every step of the reflexion analysis process. Throughout the process we frequently would investigate some dependency, realise from looking at online documentation or commit messages that we need to modify an aspect of the architecture, but then not log these changes. This made it so that when we had our final architecture only a fraction of the dependencies had an existing rationale written down. We spent a lot of valuable time having to re understand why decisions had been made. This also introduced inconsistencies in our logic which would become apparent at inconvenient times. If one member made a correct modification but no documentation was made then someone else might forget and map another component based on a false understanding of the current architecture.

The cyclic nature of the reflexion analysis was something we struggled with and we learnt that it cannot be neatly separated into different stages. Initially we wanted to split and complete each step of the process one by one however we quickly realised that this is not a viable strategy. Once we mapped the files, looking at the dependency gave us new information for errors in our mapping, and also changes to our conceptual. This would lead to a new proposed conceptual, remapping of files which upon seeing new issues would repeat this cycle. This is the Propose, Compare, Gaps, Investigate cycle shown in the following diagram from the week 7 slides,

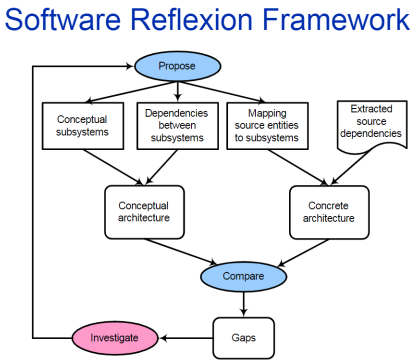

Figure 9: Reflexion Process Week 7 Slides [3, page 9]

In the future we would embrace this cyclic process, optimizing our work flow for repetition by making small changes checking how they effect the architecture and then moving on. This would allow us to continuously iterate through this process and not repeat large amounts of work.

7 Conclusions

The key takeaway from this project is that constructing the concrete architecture for a software project is far from a trivial task, especially when dealing with a relatively complex project such as Bitcoin Core. This is due to the fact that such systems can be very interconnected, exhibiting many dependencies between components. This renders increased importance to strong documentation, which becomes quite vital for an outsider to understand the inner workings of the system. By wading into this difficult system, and parsing all available documentation we were able to bring out previously proposed conceptual architecture (see A1) into line with the concrete architecture of Bitcoin

Core. Additionally by proposing a conceptual architecture for the peer to peer interface sub-component we could bring that into line with it's concrete implementation.

References

1. "Developper's Guides." *bitcoindeveloper*, https://developer.bitcoin.org/devguide/mining.html. Accessed March 22 2023.

2. Murphy, Gail C., et al. "Software Reflexion Models: Bridging the Gap between Source and High-Level Models." ACM SIGSOFT Software Engineering Notes, vol. 20, no. 4, 1995, pp. 18–28., https://doi.org/10.1145/222132.222136.

3. Week Seven Lecture Slides, Bram Adams 2023.

4. Mastering Bitcoin by Andreas M. Antonopoulos (O'Reilly). Copyright 2017 Andreas M. Antonopoulos, 978-1-491-95438-6.

5. Bowman, Ivan T., et al. "Linux as a Case Study." Proceedings of the 21st International Conference on Software Engineering, 1999, https://doi.org/10.1145/302405.302691.

6. "The Bitcoin Network." *Cypherpunks Core - Bitcoin Book*, https://cypherpunks-core.github.io/bitcoinbook/ch08.html. Accesses March 22, 2023.