CISC 322/326
Assignment 2: Concrete Architecture of Bitcoin Core
March 24, 2023

Manny Cassar	m.cassar@queensu.ca
Sawyer Proud	19sqp@queensu.ca
Cameron Krupa	19cmer@queensu.ca
Marcus Tantakoun	20mt1@queensu.ca
Duncan Scanga	19dms7@queensu.ca
Eric Jin	e.jin@queensu.ca

Table of Contents

Abstract

This report examines the concrete architecture of the Bitcoin Core. After a brief summary of previous work (from Assignment 1), we revised our previous conceptual architecture by adding the P2P Network Module and the User Interface Module, and their relevant dependencies. We then used the Understand tool to derive the dependencies of between the Bitcoin Core's Subsystems, ultimately leading to the derivation of the concrete architecture. This is followed by a reflexion analysis, which identifies important differences between the concrete architecture and our initially proposed conceptual architecture. We then move on to examine a specific subsystem in more detail – namely, the wallet module. The concrete architecture of the perception module is derived (using Understand) and reflexion analysis is again performed to highlight concrete/conceptual disparities. We then presented the two use cases - A customer wants to complete a purchase (transaction) using bitcoin and A wallet owner wants to check their wallet's bitcoin balance. In closing, the report ends with a brief discussion of the lessons learned.

Introduction

In report A1, we analyzed the conceptual architecture of Bitcoin Core, the software serving as an implementation of the Bitcoin protocol. Our analysis was mainly based on the project's documentation, online papers, and forums describing its workings. At this point, we relied on the developers' idealized conceptual framework and had not yet dug into the actual source code.

In this report, we move on to explore the concrete architecture of Bitcoin Core through analysis of its structure from the open-source code repository in great detail. Using the software tool Understand, we will analyze Bitcoin Core's software dependency tree to revise our first-suggested conceptual architecture and highlight any modifications made. We then conduct a reflexion analysis by examining the absences, convergences, and divergences between our conceptual and concrete architectures. Next, we delve deeper into the specific design of the Wallet subsystem and use two use cases: a customer completing a transaction using Bitcoin and a wallet owner checking their balance to demonstrate the inner workings of the software. Finally, we present the lessons we've learned throughout the process.

Revised Conceptual Architecture

Architectural Overview

We have made several changes to our conceptual architecture for Bitcoin Core relative to the original version from Assignment 1. Our original conceptual architecture contained several modules, including (1) RPC, (2) Wallet, (3) Storage Engine, (4) Connection Manager, (5) Peer Discovery, (6) Miner, (7) Mempool, and (8) Validation Engine. We initially did not include the P2P Network module and the User Interface module, as it was portrayed (in the Bitcoin documentation) as being part of the implementation rather than the architecture. Upon further inspection, however, it appears that the P2P Network and User Interface modules are indeed an important part of the system, facilitating communication between nodes and enabling the dissemination of new blocks and transactions. As such, we have decided to update our conceptual architecture to include these two modules, in addition to the modules listed above.

Our new conceptual architecture, with the P2P Network module included, is as follows:

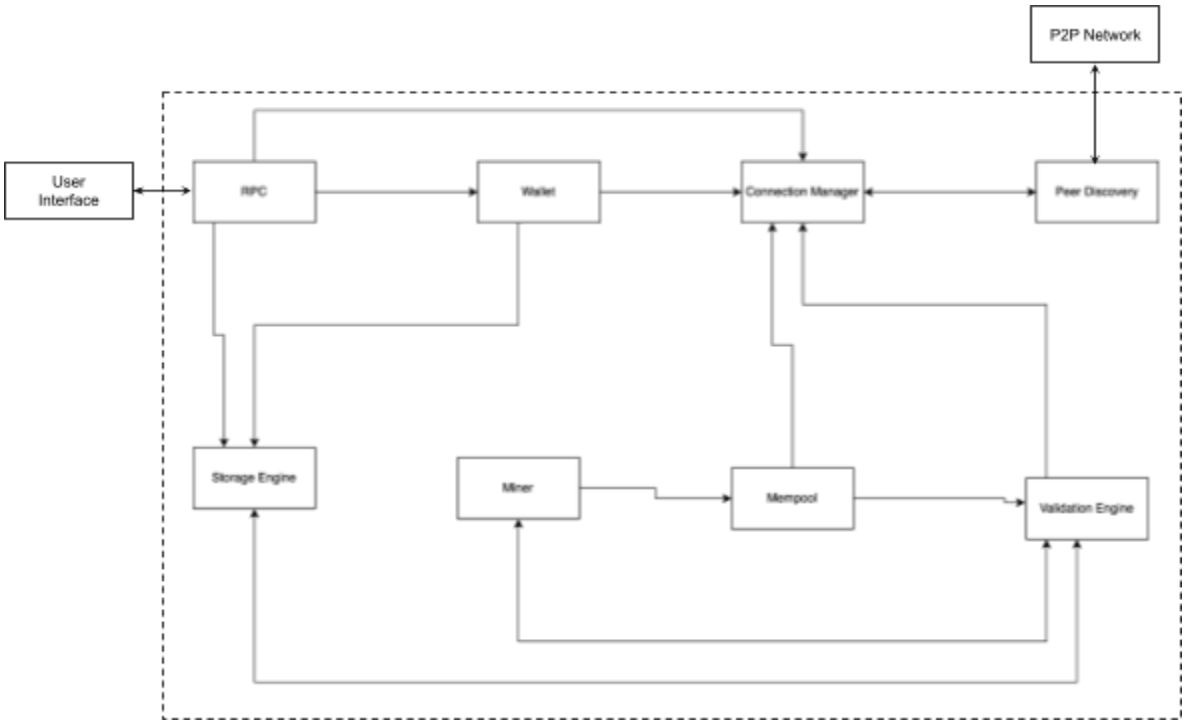

Figure 1: Box-and-Arrows Diagram of the Components

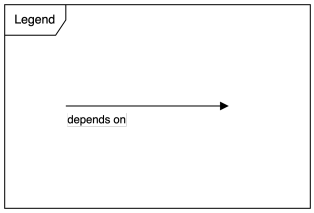

Figure 2: Legend for Box-and-Arrows Diagram

After adding the new modules, we added their relevant dependencies. First, we have added a dependency between Peer Discovery and P2P Network. This reflects the fact that the peer discovery module relies on P2P Networking to communicate with other nodes on the network. Second, we have added a dependency between RPC and User Interface. The User Interface is responsible for providing a graphical user interface to interact with the Bitcoin Core software, including options for configuring the software, viewing transactions, and managing the wallet. The Bitcoin Core software provides both a graphical user interface (GUI) and a command-line interface that is accessible through the Remote Procedure Call (RPC) protocol. The GUI module depends on the RPC module to communicate with the Bitcoin Core software and retrieve information from it, thus the dependency between the two.

Derivation Process

Using *Understand*, we can derive the concrete architecture based on the Bitcoin Core system analyzing dependencies between Bitcoin's subsystems. Using the conceptual architecture this allowed us to map Bitcoin into modules which we can then derive using dependencies. The graphing features allowed us to graph the dependencies between different modules, allowing comprehension of the system's collaboration. In Bitcoin Core, we have either one-way dependencies or two-way dependencies between the modules as seen below.

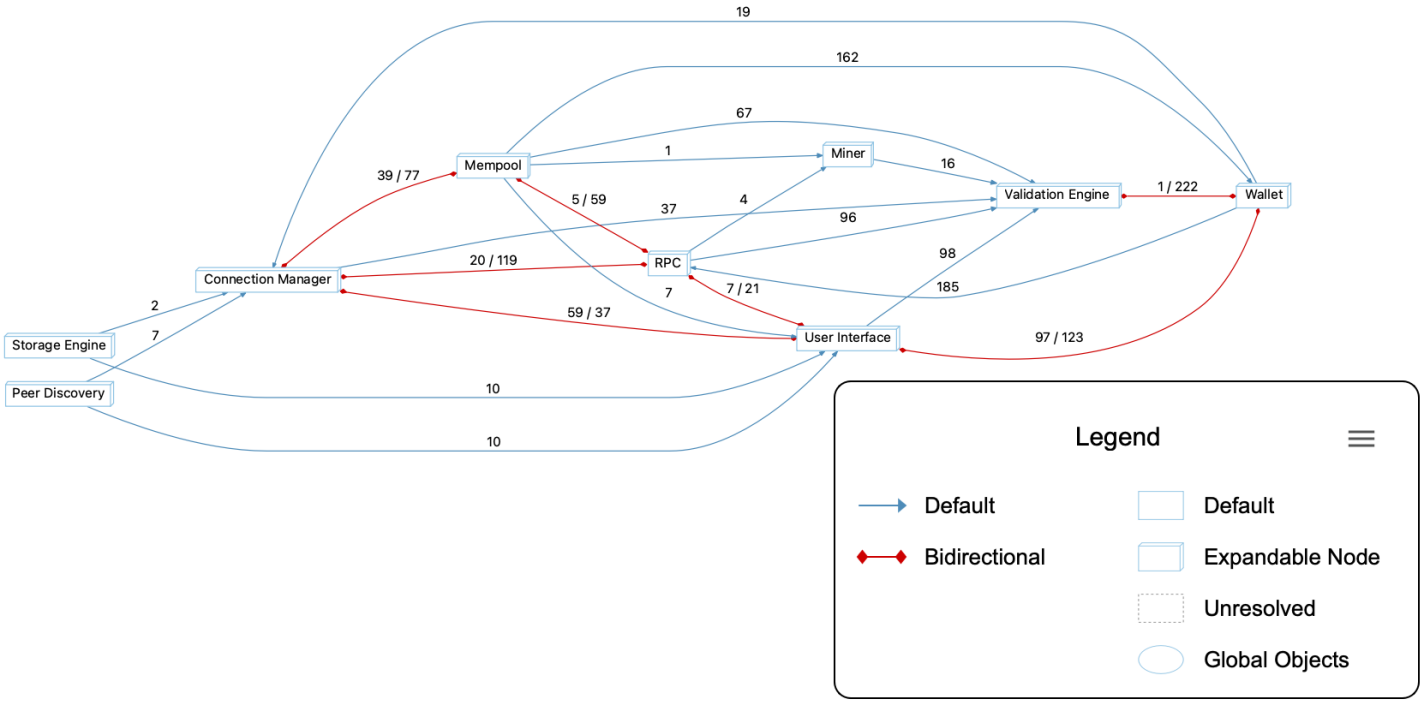

Figure 3: Dependency Graph from *Understand*

Bitcoin Core's system has many different sub-components that use dependencies allowing *Understand* to find them by creating a new architecture. Based on the new architecture, we started off by using our conceptual architecture modules which allowed us to start connecting the source code with each individual aspect. We had to understand the different components of a P2P network to determine what would go where and allow *Understand* to determine dependencies. Our dependency relation can be seen in two different ways from the graph above: single dependency (when a module depends on another module to perform tasks at hand) or two-way dependency (when both modules depend on each other for the project's outcome). *Understand* has allowed us the opportunity to see how the code can be analyzed through its dependencies which allows us to arrive at our final concrete architecture.

Final Concrete Architecture

Based on the derivation process described above, which combines static dependencies and P2P relations, we arrived at the following concrete architecture:

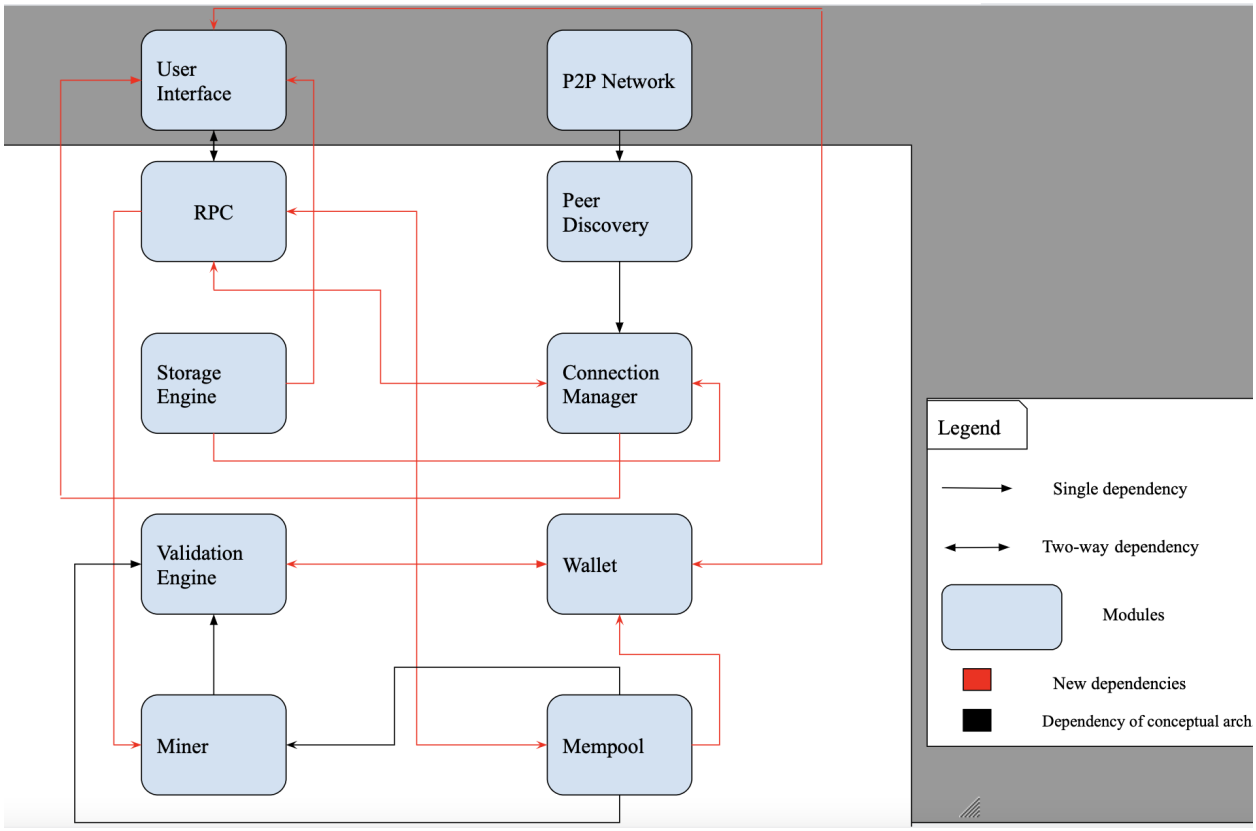

Figure 4: Concrete Dependency Graph combining Understand and P2P traffic

Our concrete architecture contains new dependencies that we have not seen in the conceptual architecture, we used the colour red to demonstrate them. We want to note that there was a dependency that has been lost from the storage engine to the validation engine. However, the rest of the dependencies that were in the conceptual architecture are also found in the figure above. The new dependencies will be analyzed in the reflexion analysis, on their meaning in the context of Bitcoin.

Reflexion Analysis

In this section, we outline the new dependencies and provide possible explanations for why they appear in the concrete architecture. Arrows from A → B indicated A depends on B. We have excluded in-depth details of dependencies concerning the newly added components (i.e. User Interface and P2P Network).

User Interface ↔ Wallet

In this two-way dependency we see a file named receivecoinsdialog.cpp that is part of the user interface and that calls the wallet to receive wallet information. However, since the user interface needs the wallet this will then become a two-way dependency to allow the network to work at full functionality.

Connection Manager → User Interface

In this dependency we see that the user interface in the file interfaces.cpp is dependent on the connection manager because it needs a handler to manage its resources.

Storage Engine → User Interface

Since the storage engine includes nodes and that the user interface needs nodes to transfer the bitcoins from one user to another. This shows how the user interface file bitcoin-gui.cpp is dependent on nodes which can be found in the storage engine.

User Interface ↔ RPC

Since the RPC needs a server in the file rpcconsole.cpp this indicates that they are dependent on one another in order to provide a server in the RPC console area.

RPC → Miner

Mining is dependent on RPC since in the file mining.cpp we need to update time and regenerate commitments which are solely done by the RPC.

RPC ↔ Connection Manager

Since the connection manager connects the different interfaces from one to another in the file interfaces.cpp, we would then need the RPCTimerInterface that provides accurate timing for the measures of bitcoin to be connected to the corresponding client. Therefore, since these are hand in hand, they are dependent on each other to keep the system under the certain threshold of timeline between transactions.

RPC ↔ Mempool

In the file rpc_mempool.cpp, we need to include mempool.h to allow the RPC/mempool function to work in a manner that allows a ressource procedural call from the mempool method. These are dependent on each other because one without the other will not allow the rpc_mempool.cpp file to work with a combination of both methods.

Storage Engine → Connection Manager

The connection manager is dependent on the storage engine file bitcoin-gui.cpp as it needs to include context.h with nodes in order to be able to connect the bitcoins through the system.

Validation Engine ↔ Wallet

Both are dependent on each other because without a wallet we cannot validate a transaction of bitcoins. In the file spend_tests.cpp, which is a validation engine we must include amount.h which is of type wallet in order to be able to see the funds and accept the transaction. Without this, the validation would not have any use.

Mempool → Wallet

The wallet is dependent on the file wallet_create_tx.cpp as it needs to include spend.h in order to create a wallet that provides accurate information through the bitcoin pending transactions before being carried to the validation engine.

Peer Discovery → Connection Manager

The connection manager is dependent on peer discovery because without any information on buyers the user would never be able to sell their bitcoins. The file bitcoin-node.cpp needs to include peer discovery which is included under context.h allowing the seller to find a buyer that is willing to do a transaction.

Second-Level Subsystem: Wallet

The wallet subsystem is how the Bitcoin Core manages and displays a user's digital currency and facilitates transactions on the bitcoin network. It interacts with the bitcoin network to retrieve transaction data and provides a secure environment for a user to manage their funds. The wallet subsystem is critical for users to participate in the network's peer-to-peer transactions.

Architecture of Wallet

The wallet subsystem uses the layered architecture style. The layered architecture works as a hierarchy with 3 tier, Wallet application, Wallet logic and Wallet data, these tier communicate between each other to satisfy the users needs. For example if a user wishes to do a transaction they will begin by interacting with the interface (Wallet application) which in turn communicates with the transaction manager (Wallet logic) that a transaction wants to be done and then communicates that a transaction is being done to the Database (Wallet data).

The wallets architectural design makes it simpler to manage, scale and modify the wallet subsystem. For instance, just the wallet application layer needs to be changed if the user wants to alter their interface. When this is done the other layers can stay the same. By doing this, the possibility of adding bugs or problems to the system is significantly decreased.

Wallet Reflexion Analysis

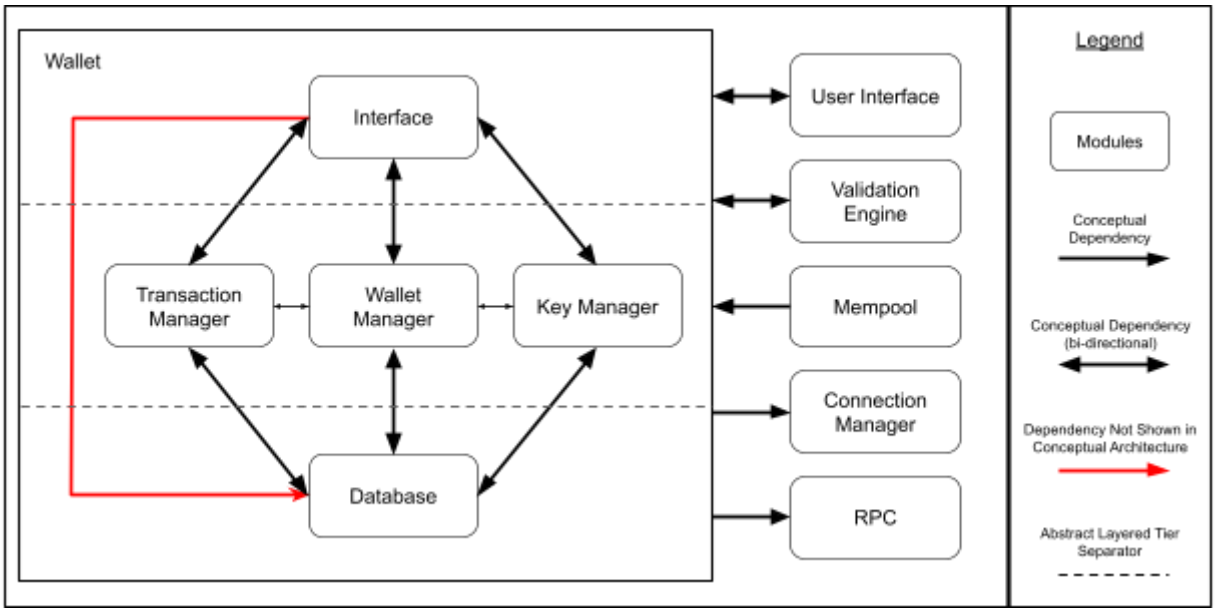

Figure 5: Discrepancies between the Wallet Component's Conceptual and Concrete Architectures

Discrepancies

Interface → Database

The Interface submodule depends on the Database submodule. The Interface is responsible for providing an interface for the user to interact with the Wallet. This includes features such as creating and signing transactions, viewing transactions history, and managing private keys. The Database submodule is responsible for storing the user's transaction history, as well as their private keys. In order for the Interface submodule to provide the user with transaction history and private key management features, it relies on the Database to retrieve this information.

In a layered architectural style, each layer provides service to the layer above it and serves as a client to the layer below. Only carefully selected procedures from inner layers are made available (exported) to the adjacent outer layers which can be uncommon depending on the system – due to violating the separations of concern. In this case, the Interface submodule (belonging to the presentation tier) shares a dependent relation directly with the Database submodule (belonging to the data tier) instead of having an intermediate (business/logic layer) to modularize these two components.

Use Cases

Use Case 1: A customer wants to complete a purchase (transaction) using bitcoin

The first use case assumes that the user (a customer) already has an account and all necessary software installed on their local machine. This use case is meant to outline the process of a normal, successful transaction. The process begins with the user scanning a QR that contains the necessary information about the requested payment. The App component, which is how the user interacts with the system, will connect to the RPC component, first establishing the connection to the network of peers through the Connection Manager. The App component communicates with the RPC component through the 'SendCoinsDialog' file to ensure the user is trying to complete a transaction. Once the connection is established through the 'Net' file inside the RPC directory, the RPC component can begin the procedures to complete the transaction.

It determines the transaction details and then asks the user if they want to confirm the transaction amount through the 'SendConfirmationDialog.' The process will be halted if the user declines the transaction amount or details. Once the user has agreed to complete the transaction, the RPC will send out a procedure to send the transaction by using the 'RawTransaction' file to communicate with the Wallet component. To accomplish this, the Wallet component will be required to ensure a UTXO of the correct value and determine the implicit fee paid to the Miner(s). Once a block is found by the Storage Engine using the 'CoinSelection' file, the RPC will then communicate with the Connection Manager component to process this selected block by finding the surrounding peers. Although not shown in the diagram to avoid confusion, if a block cannot be found or the amount requested exceeds the user's current balance, an error will be reported to the user, and the process will be halted. The Connection Manager will communicate with the Peer Discovery component to find peers as they will be required to validate and complete the transaction accomplished by broadcasting it. The Connection Manager will then work with the Validation Engine to ensure the transaction is valid by ensuring that each block read from the disk is valid. This process requires Miner components to send the data across the network and ensure no violations by registering the transaction in the 'ValidationEngine' file. Finally, the Mempool is used by the Miner component to ensure valid transactions are being combined into blocks. The Miner component uses the 'BlockAssembler' function to ensure the blocks are created with valid transactions. If the transaction is deemed invalid, an error message will be reported to the user, and the process will be halted with no change in the sender or receiver's balances. Once the transaction has been completed, and the transaction has been recorded across the network, the process will have been completed.

In the Conceptual Architecture, a dependency between the RPC and Wallet components existed to update the wallet's balance after the transaction was completed. After investigating the source code, no function calls were found to achieve this functionality. Instead, the Concrete Architecture shows that no function call is required because the user's balance is calculated based on the blocks in the Mempool component. Furthermore, once the Mempool is updated with the valid transactions, the user's balance will be updated to reflect the changes without any additional dependencies between the RPC and Wallet components.

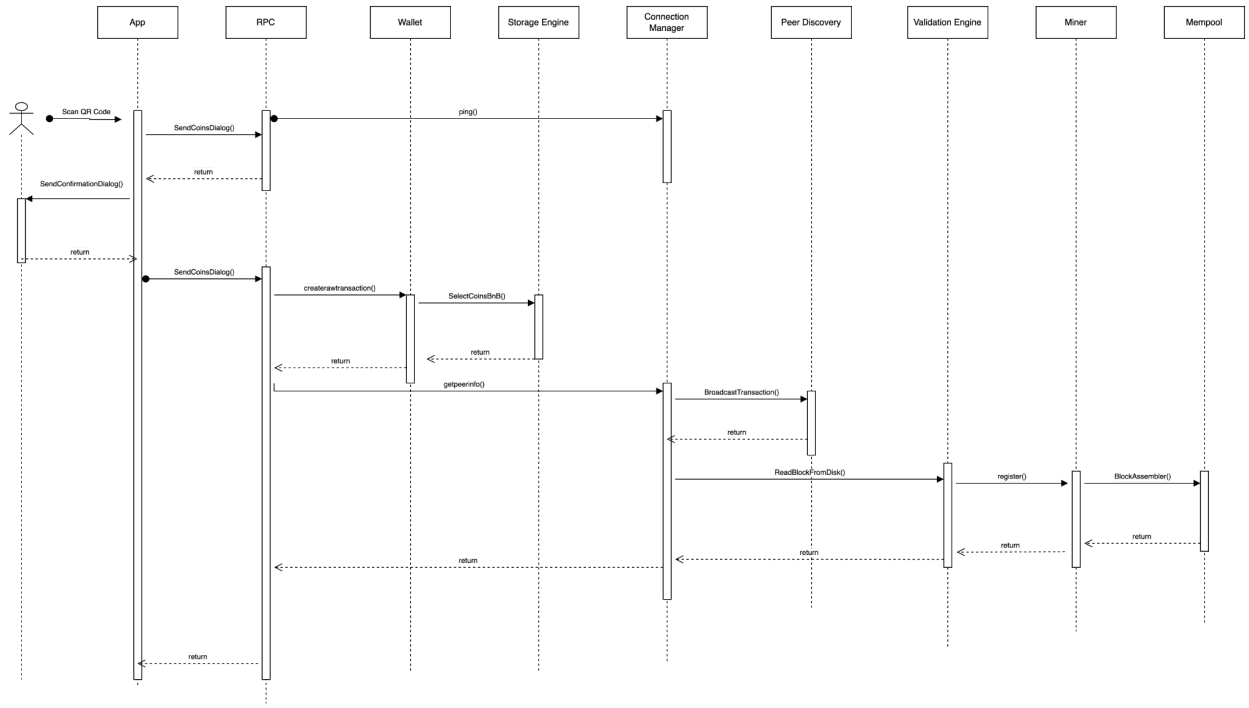

Figure 6: Sequence Diagram of Completing a Transaction

Use Case 2: A wallet owner wants to check their wallet's bitcoin balance

The second use case assumes that the user already has a working version of Bitcoin Core locally installed on their machine which is connected to a wallet. This use case is meant to outline the process of a user checking their wallet's Bitcoin balance through the Bitcoin Core GUI application.

The process starts when the user clicks on the bitcoind executable to open the App component's GUI (client) in order to reveal their Bitcoin balance. The app component initializes necessary data structures and components by calling the function AppInitMain(). Next, the App component talks to the Wallet component through its GUI interface, and runs a LoadWallet() function. This tells the Storage Engine subcomponent to return a CWallet object representing the wallet which is stored on the hard drive. From here, the Wallet component calls the function GetBalance() on the object which returns a satoshi amount stored in a CAmount object. Now that the wallet amount is loaded, we must verify it matches with the expected wallet CAmount as determined by the public blockchain. To accomplish CAmount verification, the Connection Manager subcomponent is first activated upon client instantiation where it establishes peer connections through a Discover() call to the Peer Discovery component. At this point, the Validation Engine can call it with GetNodeStats() to get to work validating new blocks and transactions in order to update the client's view of the blockchain to its current state. This updating is done with a RescanWalletTxns() call from the Wallet Component to the Validation Engine by calculating the expected CAmount as indicated through transaction history and

comparing it with the one passed by the Wallet component. Once the correct CAmount is established, its corresponding CWallet must be placed back onto the hard disk by the Storage Engine Component using the call WriteToDisk() to save locally for later offline use. Finally, the correct Wallet balance can be propagated back to the user's client through the App component GUI interface.

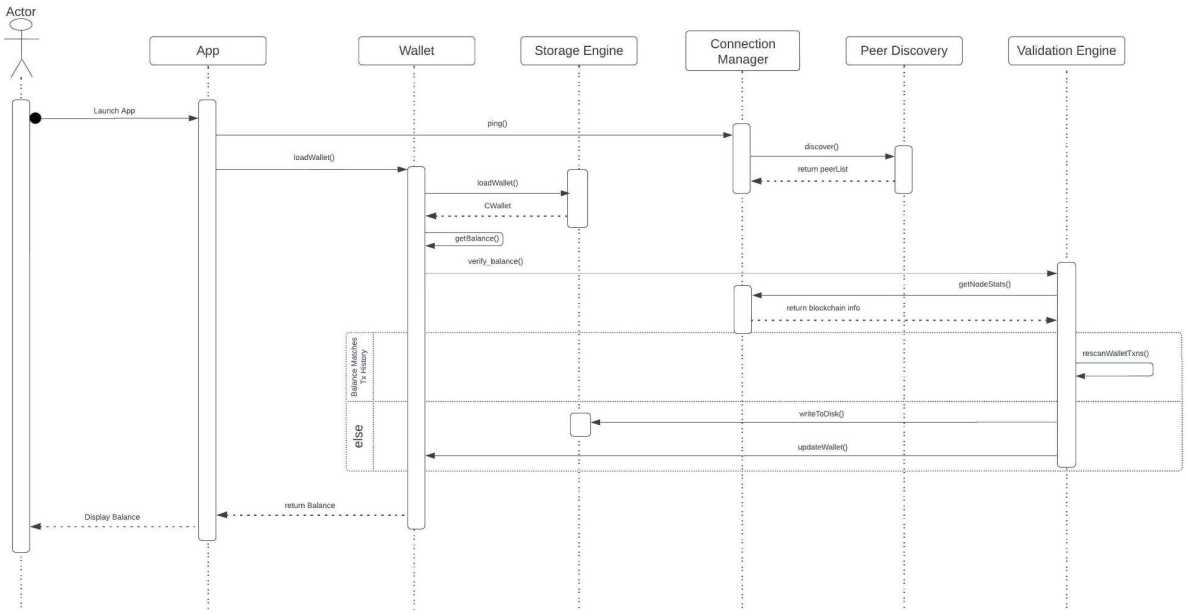

Figure 7: Sequence Diagram of Accessing Funds of a Wallet

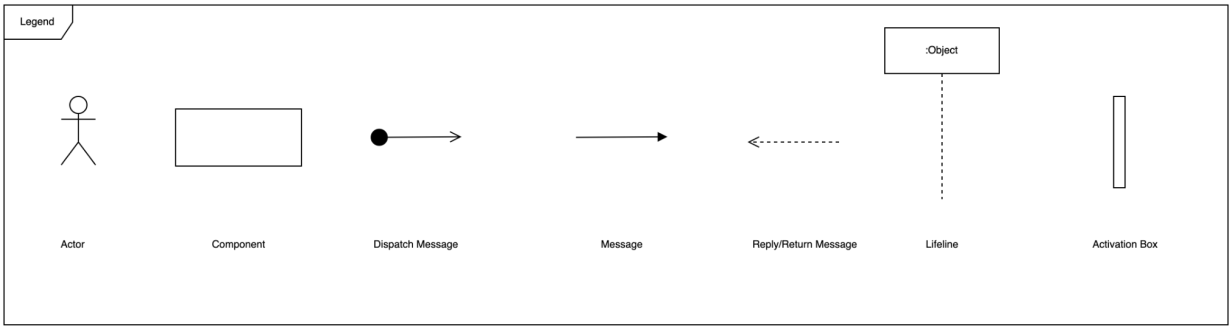

Figure 8: Legend for Sequence Diagrams

Lessons Learned

Our research into the Concrete Architecture of Bitcoin Core found that the documentation between the source code and the Conceptual Architecture diverged, specifically with the file structure and naming conventions. For example, we found a Consensus component not mentioned inside the Conceptual Architecture; however, under further investigation, this was deemed an extension of the Validation Engine component. Similarly, when designing the sequence diagrams and looking for function calls between components, the flow between files

differed from what was previously expected. Since components were spread across files and directories, we found that, in practice, the Conceptual Architecture does not always map one-to-one to the Concrete Architecture. Furthermore, through this process, we appreciated the desire to have a Concrete Architecture when working with or maintaining any source code because of how different the Conceptual and Concrete Architectures can be. With well-documented code and a valid Concrete Architecture, the time taken to make changes and fix bugs will be minimized as no time will be spent trying to understand how the system works before updating it.

Conclusion

In this report, we investigated the Concrete Architecture of Bitcoin Core. This architecture was derived using the Understand tool to derive the dependencies between the system's components/subsystems. We then compared our initial Conceptual Architecture derived from A1 with the Concrete Architecture based on the open-source code repository. Through reflexion analysis, we identified convergences, divergences, and absences between the two architectures. An in-depth analysis was performed on the Wallet component where an Architectural Style was proposed, in this case, a Layered architecture. Then the Understand tool was once again used to discern the discrepancies between the Conceptual and Concrete architectures. We then investigated the Wallet component and its relation to the other components by outlining two possible use cases and the series of dependency links through two sequence diagrams. These sequence diagrams were updated versions of the sequence diagrams described in A1 based on the source code; however, the two outlined in this report demonstrate the function calls being made between components based on the source code.

Data Dictionary

Term	**Definition**
Blockchain	Collection of Blocks stacked on top of eachother
Blocks	Set of Bitcoin Transactions
Consensus	The name used by the source code for the ‘Validation Engine’ component
Node	Computer connected to other computers, following rules and sharing information
Understand	Software tool used for code analysis
Unspent Transaction Output	Amount of digital currency authorized by one account to be spent by another

Naming Conventions

Acronym	**Full Context**
A1	Assignment 1 (Conceptual Architecture)
API	Application Program Interface
CLI	Command Line Interface
OS	Operating System
P2P	Peer-to-Peer
QR Code	Quick Response Code
RBF	Replace-By-Fee
RPC	Remote Procedure Call
UI	User Interface
UTXO	Unspent Transaction Output

References

BitStamp. (2022, August 17). *What is a bitcoin improvement proposal? (BIP) - bitstamp learn center*. What is a Bitcoin Improvement Proposal? (BIP). Retrieved February 16, 2023, from https://www.bitstamp.net/learn/blockchain/what-is-a-bitcoin-improvement-proposal-bip/

Bitcoin. (2016). *Bitcoin/Bitcoin: Bitcoin Core Integration/Staging tree*. GitHub. Retrieved March 19, 2023, from https://github.com/bitcoin/bitcoin

Bitcoin Core. (2019). Version History. Bitcoin.org. Retrieved February 15, 2023, from https://bitcoin.org/en/version-history

Nakamoto, S. (2002). *Open source P2P money*. Bitcoin. Retrieved February 15, 2023, from https://bitcoin.org/bitcoin%20pdf

Saini, D. (2013, December). *Security concerns of object oriented software architectures - researchgate*. Security Concerns of Object Oriented Software Architectures. Retrieved February 15, 2023, from https://www.researchgate.net/publication/259470705_Security_Concerns_of_Object_Oriented_Software_Architectures

Team, B. (2011, September 15). *Lan 101: Networking basics*. Tom's Hardware. R. 02/15/2023, https://www.tomshardware.com/reviews/local-area-network-wi-fi-wireless,3020-2.html

Unknown. (2017). *Object Oriented Architecture* . Object oriented architecture. 02/15/2023, from https://www.tutorialride.com/software-architecture-and-design/object-oriented-architecture.htm

Unknown, U. (2009). *Bitcoin Core Validation*. Validation - Bitcoin Core Features. Retrieved February 15, 2023, from https://bitcoin.org/en/bitcoin-core/features/validation