CISC 322/326 - Software Architecture
Assignment #2: Report
Friday, March 24, 2023

Concrete Architecture of Bitcoin Core

Team Bingus II
Connor Decan - 19ctd3@queensu.ca
Allen Geng - 20lg5@queensu.ca
Kevin Jiang - 19shj1@queensu.ca
Yuanqi Liang - 18yl204@queensu.ca
Jayden Ting - 19tkt1@queensu.ca
Kevin Zhang - 19jyz5@queensu.ca

Abstract:
This report will cover the concrete architecture of Bitcoin Core and its Wallet subsystem. We used Understand to derive the main concrete architecture and find differences between our conceptual architecture and how it was actually implemented. We also provide two use case sequence diagrams to show how parts of the system work. Finally, we give our conclusions and findings, alongside the lessons learned from completing this task.

Introduction and Overview:
The emergence of Bitcoin Core has transformed the way people transact in the digital world. As the adoption of bitcoins as a mode of payment continues to rise, it has become increasingly important to understand the complexities of the Bitcoin Core architecture to ensure the smooth functioning of the system. Bitcoin Core's decentralized and open-source software has made it a popular choice for users, but its intricate design requires a thorough analysis of its various subsystems to fully comprehend the system's functionality.

This report aims to delve into the architecture of Bitcoin Core, particularly the Wallet subsystem, utilizing the powerful Understand tool by SciTools to derive the main architecture and identify differences between the conceptual and concrete architecture. By doing so, readers will gain a comprehensive understanding of the architecture of Bitcoin Core and its Wallet subsystem, which will prove invaluable in comprehending the system's functionality and making informed decisions regarding its use. In addition, this report will explore the potential benefits and drawbacks of using Bitcoin Core, providing readers with a well-rounded view of the system and its place in the digital economy.

Architecture:

Derivation Process:

To derive the concrete architecture for Bitcoin Core, we used the tool provided to us, Understand by SciTools. We took the information that we had from our previous assignment about the conceptual architecture and applied it to the app to try and see how the subsystems interacted. First, we generated a dependency graph for the folders in the filesystem to see how they interacted with each other. Unfortunately, it seems like the folders are not fully representative of the actual system architecture, so we used another method. We made an architecture based on our conceptual architecture, added some new subsystems that we found based on the code such as a logging system, and then tried to map each file to its respective subsystem as shown in the tutorial we were given. Unfortunately, due to the confusing nature of the given files and the almost complete lack of documentation, it was difficult to accurately determine which subsystem some of the files belong to. While some of them were more obvious, such as how the “wallet” files very likely mapped to the wallet subsystem, others were not so clear, especially since many are not organized into corresponding folders like the rest. Regardless, we used the information we had to try and come up with an accurate concrete

architecture. For any divergence, or new subsystems, we investigated their rationale by examining the source code, github commit history, and release note for the application. This source of information gives us an understanding on what each side of the dependencies is responsible for, why they were added, and any new features that were later implemented.

Concrete Architecture:

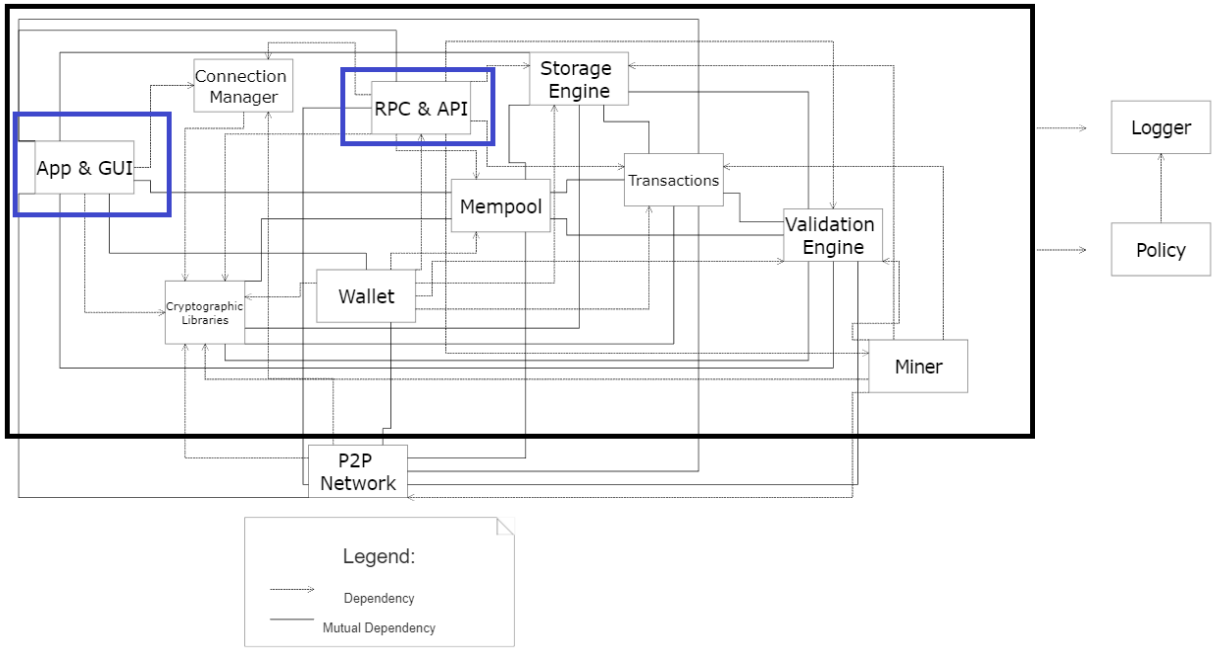

Figure 1: Concrete Architecture

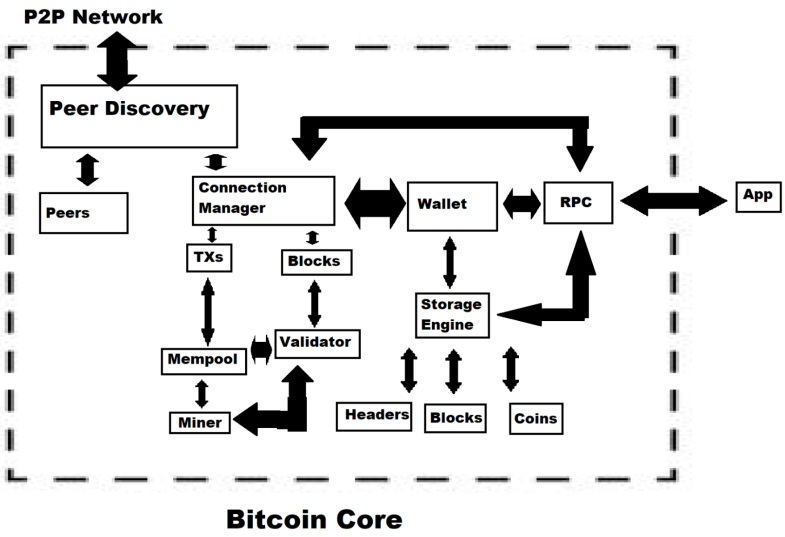

Figure 2: Conceptual Architecture (For reference)

Architectural Style
Bitcoin Core is built around a peer to peer style architecture. It uses this to allow the Bitcoin nodes to connect and send messages to each other. This is the main method of data sharing used in Bitcoin Core. It can be seen in figure 1 through the P2P Network subsystem. It also uses a publish-subscribe style and a client-server style. The client that Bitcoin Core uses is called the qt protocol, and it serves as the main way of accessing a GUI. The pub-sub style is used in the RPC & API subsystem, under the ZMQ interface.

Top-Level Concrete Subsystems:

App & GUI:

src/qt:
This subsystem is the main client for Bitcoin Core, and it also allows users to access a GUI frontend. It interacts with a number of subsystems that need access to the user interface. The RPC system is accessed through an rpc console in the UI as shown in qt/rpcconsole.cpp. The wallet subsystem is accessed to give users controls as seen in coincontroldialog.cpp. The UI accesses the P2P Network as seen in clientmodel.cpp, etc.

Connection Manager:

src/ipc:
This subsystem is what allows users to connect to the broadcast network so that they can create and communicate transactions and blocks. It interacts with the cryptographic libraries to use the functions provided, such as how in src/protocol.cpp they need to modify attributes of threads.

RPC & API:

src/rpc:
This subsystem provides a server that allows users to access the blockchain and make transactions using a publish subscribe architectural style. It interacts with the connection manager when rpc/node.cpp needs to access the ipc system to spawn a new bitcoin node. Although the function comments state that it is meant for testing, it still accesses the other subsystem. It interacts with the policy subsystem because the server needs access to policy info, such as how fees.cpp needs access to policy's feerate, etc.

src/zmq:

The zmq is the pub-sub way for bitcoin core to carry messages across ipc. It interacts with the P2P network, such as in zmqpublishnotifier.cpp where it needs to check if an address is IPV6. It also interacts with the mempool, as zmqnotificationinterface.cpp overrides the ability to add and remove transactions from the mempool. It also interacts with the validation and storage engines for a similar reason, but instead of transactions it connects and disconnects blocks in the same file.

Cryptographic Libraries:

This subsystem and its subparts interact with many other subsystems, as they call on the contained functions to apply them for the other subsystems' tasks.

src/crypto:

This section contains several hashing functions such as sha256 and poly1305.

src/secp256k1:

This section contains code to generate an elliptic curve used in Bitcoin's public-key cryptography.

src/univalue:

This section contains code for a universal value object that inputs and outputs JSON data.

P2P Network:

src/node:

This subsystem provides nodes that store and relay messages to peers such as a version message, IP address, etc. It interacts with the Connection Manager as seen in bitcoin-node.cpp when it needs to access the ipc interface. It also interacts with the Storage Engine, as seen in net_processing.cpp when it checks if blocks are connected and disconnected.

Validation Engine:

src/crc32c:

The Cyclic Redundancy Check validates the data to prevent any accidental changes. It interacts with all the subsystems that store data, such as Storage Engine, Transactions, etc, by verifying the data before it is sent anywhere so no discrepancies occur.

src/consensus:

This section contains functions to validate various parts of bitcoin core, specifically transactions as seen in tx_verify.cpp and storage engine such as in consensus.h.

Wallet:

src/wallet:

This subsystem stores a collection of private keys that is used to manage those keys. It interacts with the transaction system, P2P network, and RPC & API to make transactions over the network. It also interacts with the storage engine to access key data, and the APP & GUI to display wallet data visually.

Logging:

This subsystem makes logs of all the other subsystems. It needs to interact with all of them in order to track their activity and create the logs for them.

Policy:

This subsystem includes information about the behavior of miners, nodes, etc that is meant to be set by the user. Since it is customizable, it interacts with all the other subsystems as they need to access the modified information.

Transactions:

This subsystem provides the ability to make transactions over the network. It interacts with wallet and the P2P system to make the transactions as mentioned prior. It also interacts with the mempool and validation engine such as in txmempool.cpp.

Miner:

This subsystem uses the cryptographic libraries to calculate hashes. It interacts with the P2P network and the Storage Engine as seen in node/miner.cpp as it uses a block assembler. It also interacts with the transactions as seen in the block assembler function.

Mempool:

The memory pool is responsible for storing unconfirmed transactions and other information in non-persistent memory. It interacts with many other subsystems that need to load the memory pool such as the Validation engine in validation.cpp, and wallet in wallet_create_tx.cpp

New Subsystems:

Cryptographic Libraries: Cryptographic Libraries are vital for implementing security mechanisms and hashing algorithms to create unique digital fingerprints of transactions and blocks. This is necessary to safeguard the network against any unauthorized access or tampering.

Logger: Logger offers various capabilities that aid in the efficient logging and analysis of network events. These include the ability to specify the level of detail captured in the logs and the implementation of log rotation to avoid issues with large and cumbersome log files. This allows developers to customize the logging system according to their specific requirements.

Policy: Policy sets the guidelines and parameters that control the actions of all network participants, including miners, nodes, and wallets. It mandates that all members adhere to the same regulations and policies, such as transaction fees, block size limitations, and consensus rules. Policy also allows for the customization of network policies based on specific situations or needs, allowing developers to adjust the network's behavior to achieve maximum efficiency and safety.

Reflexion Analysis for Dependencies:

New Dependencies:

Connection Manager → Cryptographic libraries

Rationale: Connection Manager is to manage connections to other nodes on the Bitcoin network, using algorithms and protocols for secure connections provided by the cryptographic library to ensure secure connections and data transfers between nodes.

Wallet → Cryptographic libraries

Rationale: Wallet is responsible for securely managing private keys and digital signatures, both of which are vital elements in the Bitcoin protocol. To perform these cryptographic operations securely and efficiently, the wallet depends on the functionality and algorithms provided by cryptographic libraries. As a result, there is a dependency from “bitcoin/src/wallet” to “bitcoin/src/wallet/crypto”.

RPC & API → Cryptographic libraries

Rationale: RPC & API rely on cryptographic libraries to ensure secure communication between different components of Bitcoin Core. These libraries provide important functions and algorithms for secure key exchange, authentication, and encryption, which are necessary for protecting data transmitted through the RPC and API. As a result, there is a dependency from “bitcoin/src/zmq/zmqpublishnotifier.cpp” to “bitcoin/src/crypto/common.h”.

Everything in P2P network → Loggers & Policy

Rationale: New modules called Policy and Logger have been created to enhance communication between different modules within the system. It serves as a library of shared resources, enabling modules to access important information generated by other modules. As a result, every module in the system relies on the Policy and Logger module, specifically from “bitcoin/src/node/” to “bitcoin/src/policy/policy.h” files, for communication purposes.

Policy → Loggers
Rationale: Loggers can be valuable sources of information that guide decision-making and ensure proper software performance, and they can be adjusted to record policy-relevant information, such as the number of transactions per block or transaction confirmations, for policy monitoring and adjustment.

Removed Dependencies:

P2P network → Peer Discovery
Peer Discovery → Peers
Rationale: Peer Discovery is temporarily disabled by default due to network security issues, ensuring that they do not negatively impact the overall operation of the software.

Connection manager ⇄ Blocks
Rationale: Since blocks are not managed directly by the connection manager, but by a dedicated blockchain module, blocks are removed from the connection manager. Separating the management of blocks from these components allows for more efficient handling of larger blockchains and a more flexible blockchain data management architecture.

Storage Engine ⇄ Headers, Blocks, Coins
Rationale: These three dependencies were removed because the storage engine split into head, block, and coin would make it more difficult to modify or extend the software between these components. Any change to one of these components may require changes to other components, which may introduce new bugs or problems. And it can add to the overall complexity of the software, making it harder to understand and maintain, test and debug.

Inner Architecture: Wallet

According to our Conceptual Architecture Report, the Inner Conceptual Architecture for the wallet consists of 2 components: the Wallet Program and the Wallet File. The Wallet Program utilizes up to two main features (distributing keys and signing) to allow the user to perform transactions through the Wallet's Interface and the Network while the Wallet File is responsible for storing and managing a user's keys and is depended on by the Wallet File.

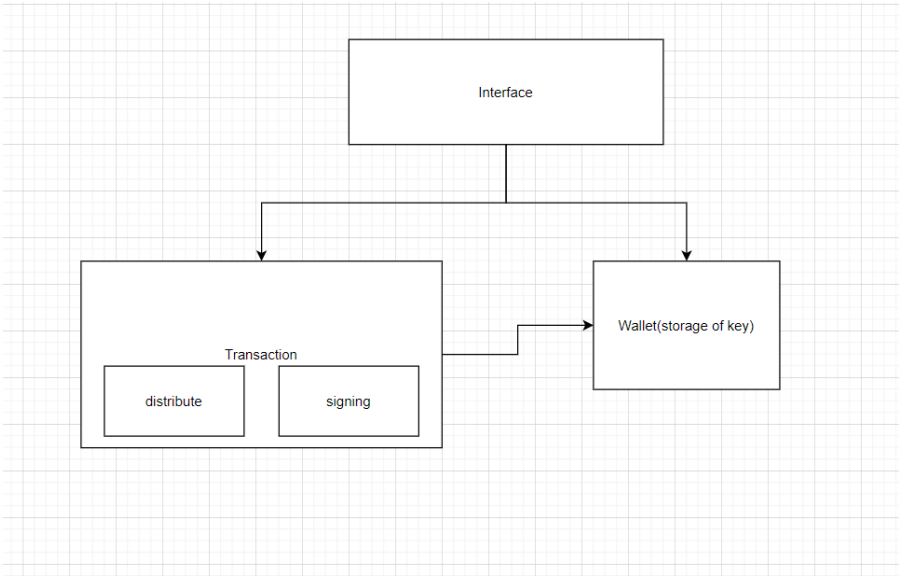

Figure 3: Inner Conceptual Architecture: Wallet

In the Concrete Architecture, there are no absences. However, it has an obvious divergence; the Wallet File depends on the Wallet Program's features to sign keys in the storage.

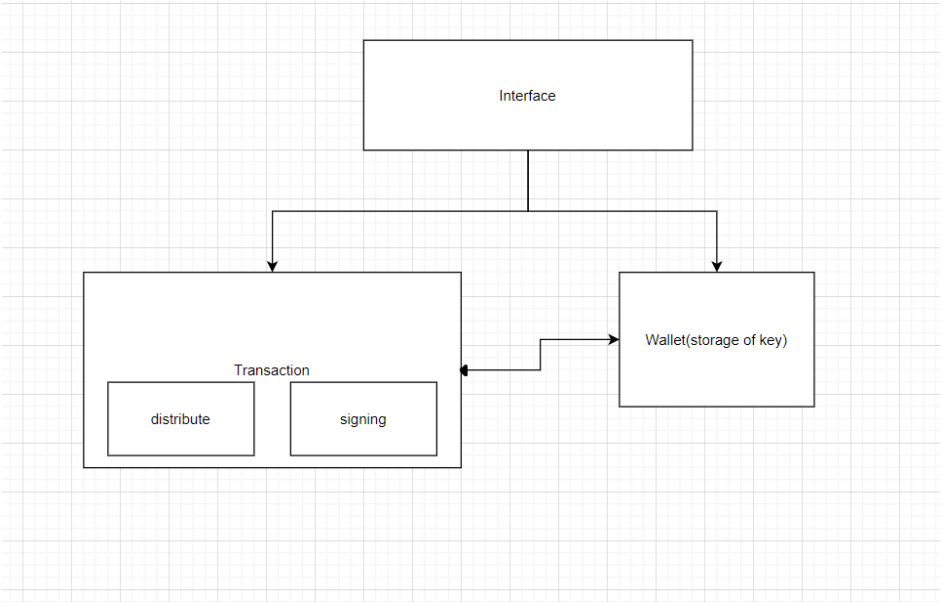

Figure 4: Inner Concrete Architecture: Wallet

Wallet Reflexion Analysis:

New Dependencies:
Wallet Program → RPC
Rationale: Originally RPC was not included in the conceptual architecture for the wallet we proposed. When examining the source code, we found that the interface depends on several files

in the rpc folder of the wallet. This allows the user to perform actions using other applications, such as a separate application for signing, network, or distribution.

Storage→Transaction
Rationale: Instead of a one way dependency, the concrete architecture shows a mutual dependency between storage and transaction, specifically for the signing component. Storage now includes an external_signer file which belongs to the wallet program. This new feature allows the use of external signers such as hardware wallets when signing a transaction.

Distribute→Fees Management
Rationale: The file feebumper.h performs some to set and validate fees on an outgoing transaction. More specifically it is dependent on the spend.h file which is responsible for distribution. As fees sometimes may be included in a transaction, which is why this dependency is needed. This component will make sure that the fees are following the rules such as a minimum or maximum. It also helps to estimate fees for the outgoing transaction based on previous records.

New component:

Coin Control system:
Conin Control system was added so that it can help the application to manage coins. It acts as a manager of the coins in a wallet. It may be responsible for selecting which coin to use for the transaction, and also set constraints on the use of coins. This component is depended on by the wallet program. Files include coincontrol.h, and coinselection.h

Fees Management:
Component specifically responsible for fee management on transaction. It may be responsible for generatinting, validating fees for outgoing transactions.

RPC & API:
RPC & API is a new component that was discovered in concrete architecture. As discussed in our first report. A single wallet application does not need to perform all three tasks: sign, distribute, and network. Having separate wallet programs helps with security as several sections can be run offline or on separate hardware for increased security. This RPC & API component implements this possibility by letting the user call the procedure on another system or application, which allows a separation of task between different wallet applications.

Use Cases:

Use Case #1: Sending Bitcoin to a Single Address

Sending Bitcoin involves two steps. First, the transaction is created by checking parameters within the wallet. Then, the transaction is broadcasted to peers.

The sequence diagram shows the generalized flow that a JSON RPC request takes to send a transaction to a single address. Omitted from the diagrams were several checks that were assumed to be valid for brevity, as well as several calls to logging made, as it can be reasonably assumed to exist and occur for the transaction.

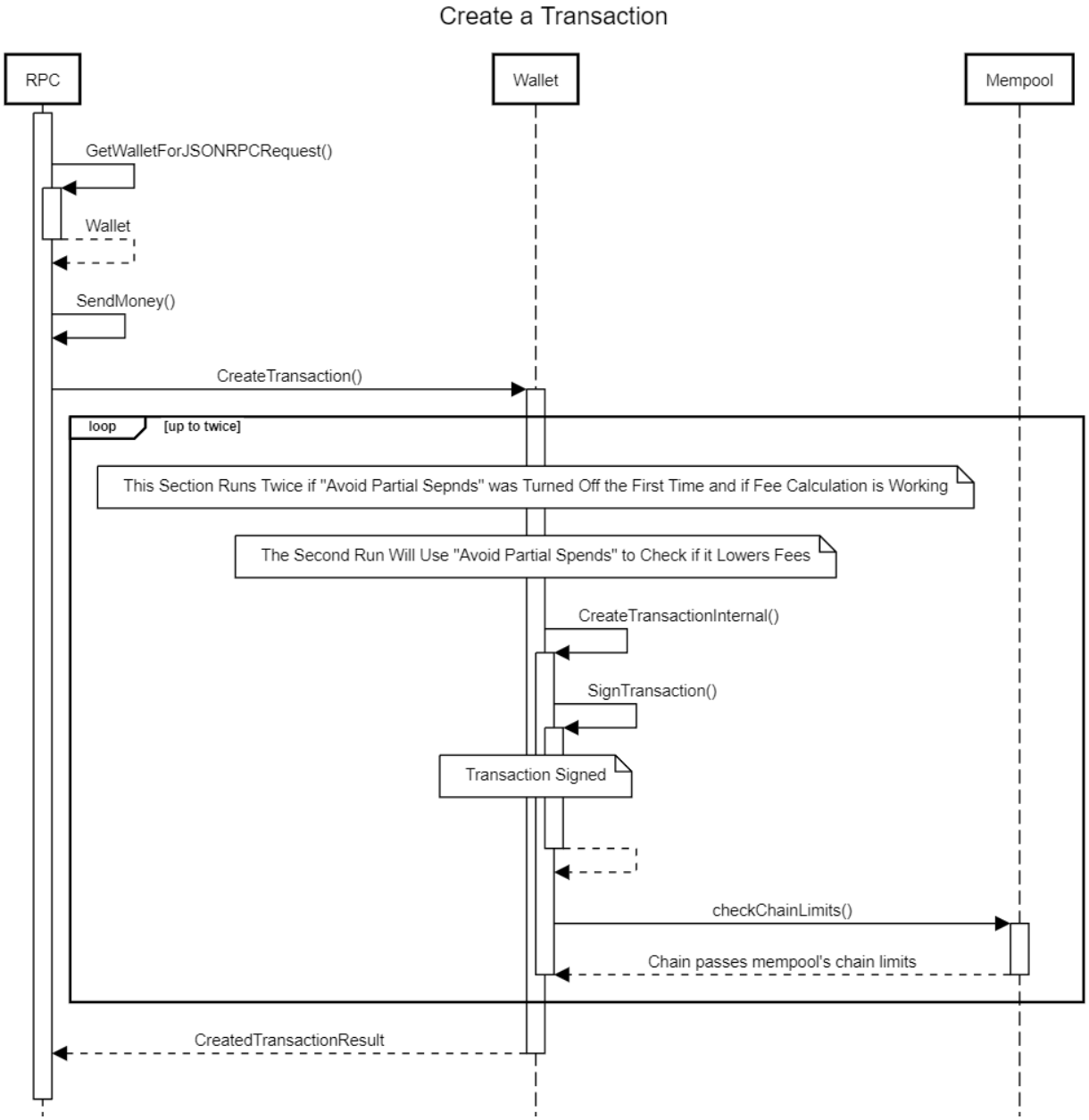

Figure 5: Creation of a Transaction

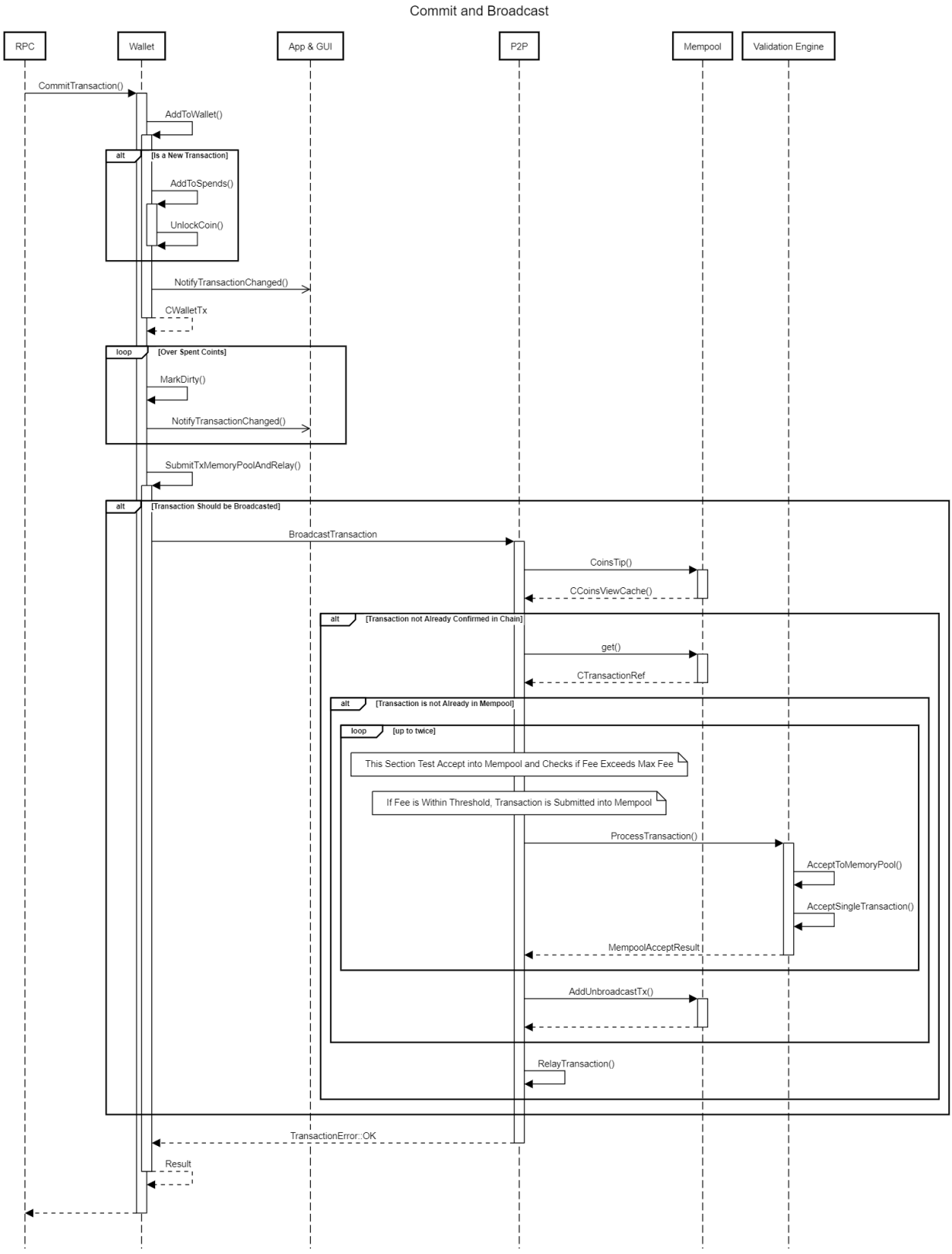

Figure 6: Broadcasting the Transaction

Use Case #2: Receiving Bitcoin

Receiving Bitcoin involves the following steps. First, the Apps & GUI subsystem receives an action and a request to receive coins. It then creates an address by calling the CreateAddresses function and getting an address from the Wallet subsystem using the GetAddress function. The Wallet subsystem then writes this data to the Storage Engine subsystem, which confirms the success of the write. The address is then displayed to the user through the Apps & GUI subsystem.

When a new transaction is received, the Wallet subsystem retrieves it by calling the GetTransaction function from the Mempool subsystem. The Mempool subsystem then checks if the transaction is valid by calling the CheckTransaction function. If it is valid, the Wallet subsystem processes the transaction by calling the ProcessTransaction function from the Mempool subsystem. The transaction status is then notified to the user through the Apps & GUI subsystem.

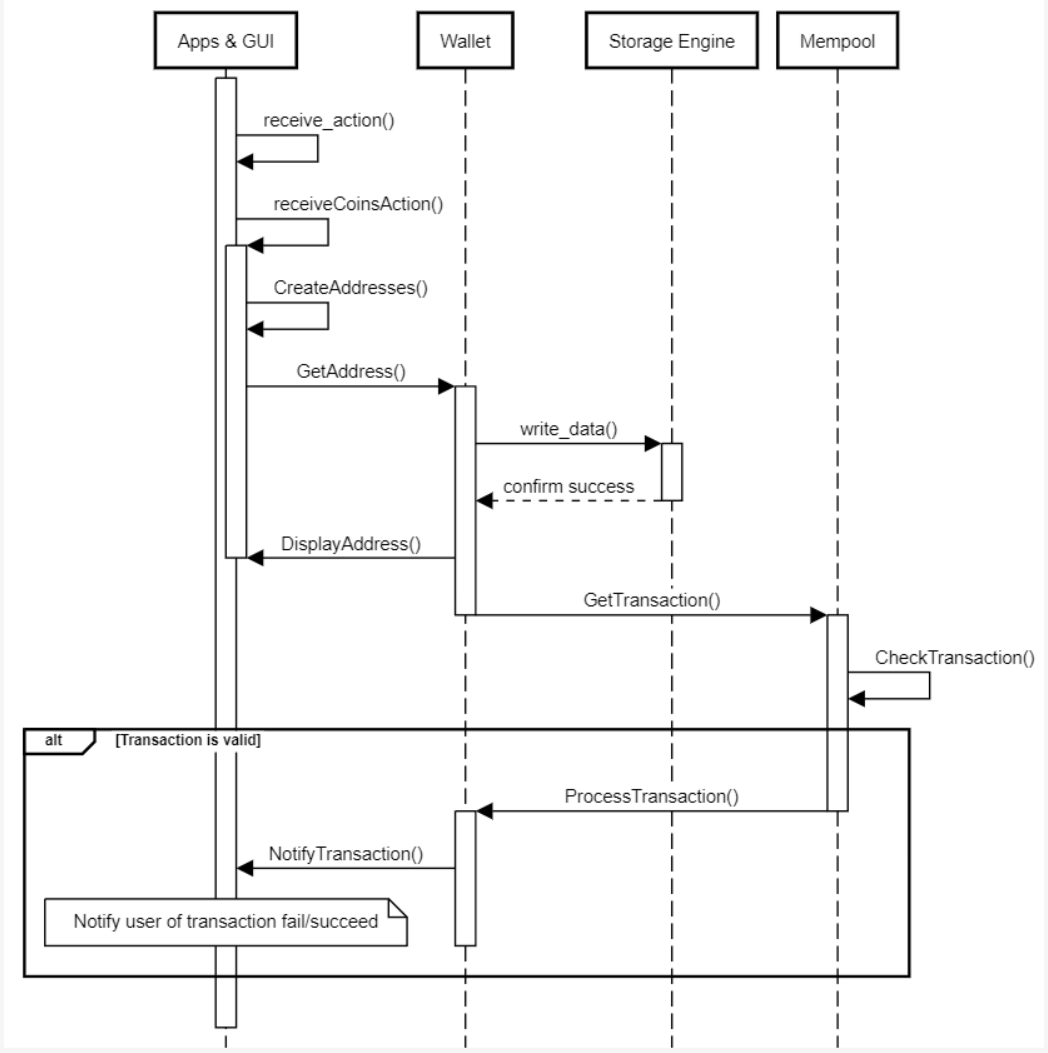

Figure 7: Receiving Bitcoin

Naming Scheme:
RPC - Remote Procedure Call
Zmq - Zero Message Queue
Ipc - Instant Partial Confirmation

Conclusions:
With a combination of our previous conceptual architecture and the Bitcoin Core source code we were able to use Understand to create a concrete architecture. We were also able to create a specific conceptual architecture for the Wallet subsystem. Using this information, we were able to analyze any differences between our created conceptual architectures, and our generated concrete architecture through a series of reflexion analyses. Finally, we were able to come up with two use cases and create diagrams to demonstrate them.

Lessons Learned:
To fully comprehend a software system's specific architecture and how its different components interact, it is essential to reference and analyze the source code. By doing so, one can create a detailed map of the system's architecture, which includes modules, functions, and dependencies. This map can then be used to analyze any differences between the intended conceptual architecture and the actual specific architecture. Additionally, examining use cases, identifying potential issues or limitations, and proposing solutions can help to refine one's understanding and ideas. Collaboration with others is also crucial in improving one's understanding of complex systems, as it enables every aspect to be refined and any previously unnoticed problems to be identified. Ultimately, this collaborative effort can help to refine our understanding of the system architecture and ensure that it is functioning optimally.

Sources:

"Bitcoin Core." Bitcoin Core - Bitcoin Wiki, https://en.bitcoin.it/wiki/Bitcoin_Core.

"SECP256K1." Secp256k1 - Bitcoin Wiki, https://en.bitcoin.it/wiki/Secp256k1.

"Libbitcoin Client." Libbitcoin Client - Bitcoin Wiki, https://en.bitcoin.it/wiki/Libbitcoin_Client.

"API Reference (JSON-RPC)." API Reference (JSON-RPC) - Bitcoin Wiki, https://en.bitcoin.it/wiki/API_reference_(JSON-RPC).

"Network." Network - Bitcoin Wiki, https://en.bitcoin.it/wiki/Network.

"BIP Draft - Instant Partial Confirmation." BIP Draft - Instant Partial Confirmation - Bitcoin Wiki, https://en.bitcoin.it/wiki/BIP_Draft_-_Instant_Partial_Confirmation.

"Poly1305." Wikipedia, Wikimedia Foundation, 10 Feb. 2023, https://en.wikipedia.org/wiki/Poly1305.

"Cyclic Redundancy Check." Wikipedia, Wikimedia Foundation, 19 Mar. 2023, https://en.wikipedia.org/wiki/Cyclic_redundancy_check.

Bitcoin. "Bitcoin/SRC at Master · Bitcoin/Bitcoin." GitHub, https://github.com/bitcoin/bitcoin/tree/master/src.

"RPC API Reference." Bitcoin, https://developer.bitcoin.org/reference/rpc/index.html.