CISC322 - Group 19 - Concrete Architecture

Ethan Kim, Arturo Lomonaco, Sean Liang, Aidan Leyne, Sebastian De Luca, Robbie Huang

Introduction and Overview

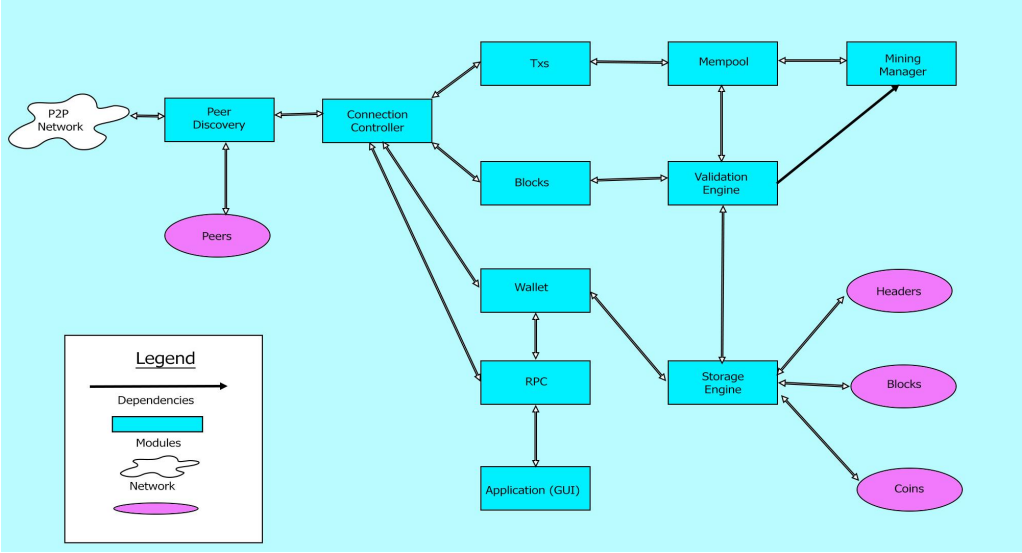
- This report aims to analyze and discuss the concrete architecture of Bitcoin Core, which uses a P2P architecture for decentralized interaction.
- It also employs Publish/Subscribe architecture for handling transactions.
- The report covers the derivation process and visualization of the concrete architecture using Understand.
- It provides a detailed description of the concrete architecture, including a review of the conceptual architecture and Reflexion Analysis, which introduces new subsystems and dependencies.
- A specific second-level subsystem for the wallet component is also introduced.
- The report updates sequence diagrams for use cases such as checking wallet balance and sending transactions.
- Finally, the report concludes with a summary of discoveries and outlines future work.

Derivation Process

- To formulate the concrete architecture for Bitcoin Core, the team used the program Understand to visualize the provided source code.
- Before using the tool, they reviewed the conceptual architecture and feedback from the last assignment.
- The unstructured nature of files and lack of hierarchical directories in the codebase made understanding the architecture difficult.
- Removing lone files and hiding non-architecture-related folders from the dependency view helped to simplify the high-level architecture.
- For the second-level rpc subsystem, the team used filenames, method names, and commit messages to distinguish the roles of each file.
- The best view of the concrete architecture is presented in the report.

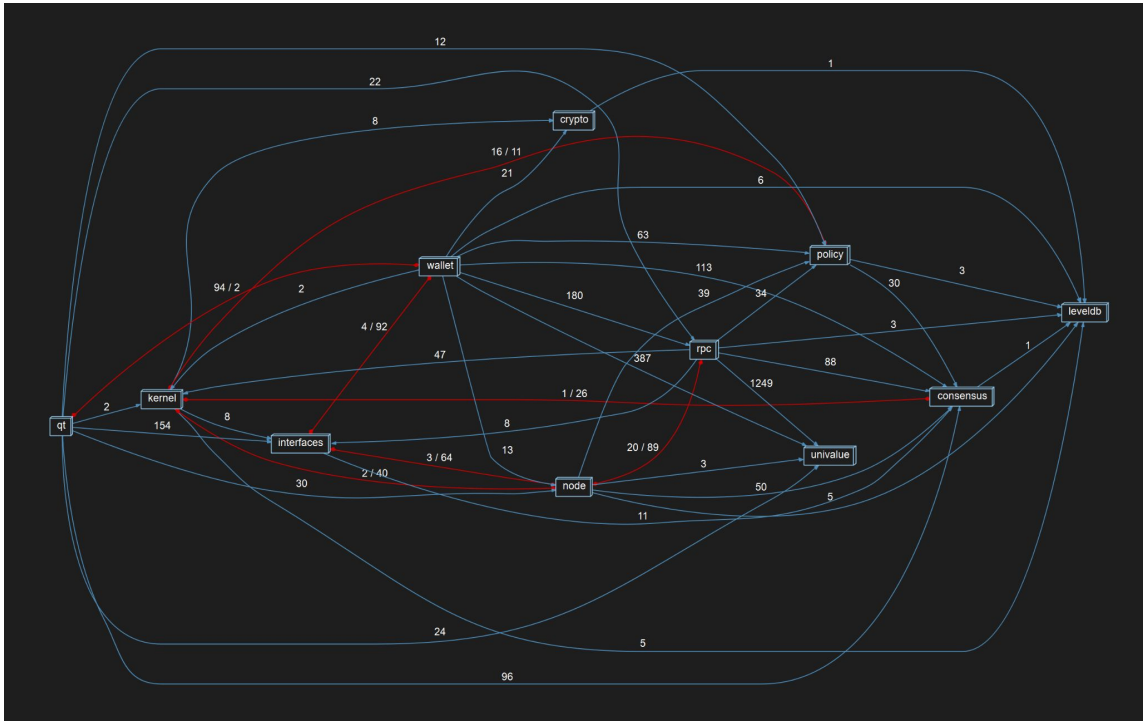
Review of Conceptual Architecture

- While the concept of Bitcoin or cryptocurrencies is based on a peer-to-peer model, the architecture for the Bitcoin Core software is publication-subscription (pub-sub).
- Each BTCC client does not need a constant connection to the chain and instead only needing to know when new information arrives.
- The software connection to the chain through the Peer Discovery module where the accrued information passes to the Connection Controller for distribution and to a database of peer information.
- The Wallet, RPC, the Blocks and Tx all communicate with the manager, where the RPC then passes on to the front-end application and the rest of the modules take care of all processing and core functionality.
- Blocks validate through the Validation Engine and then distributed and stored in their respective databases.



Concrete Architecture

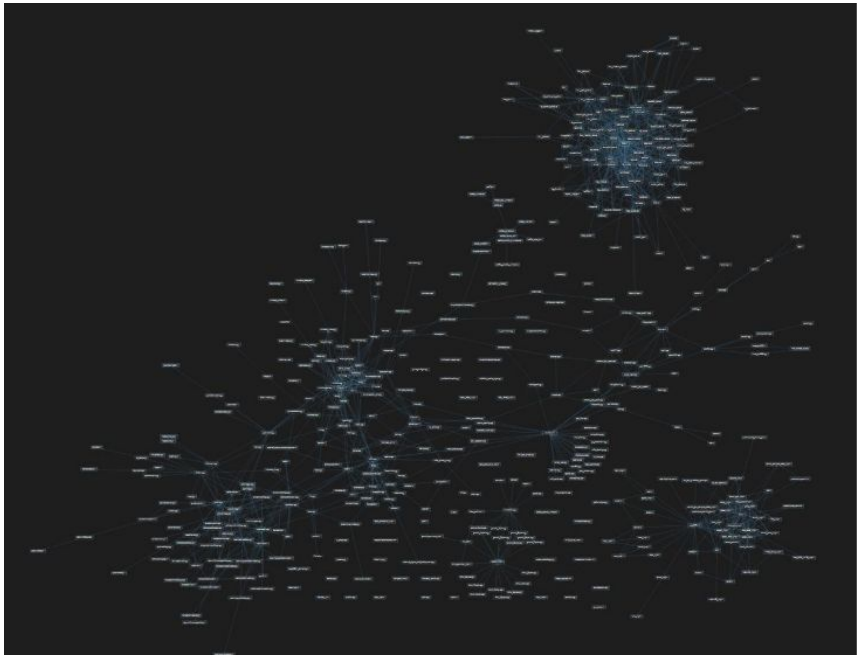
- This is the main diagram that we used for our concrete architecture.
- On the next page will be a more detailed version



Concrete Architecture Continued...

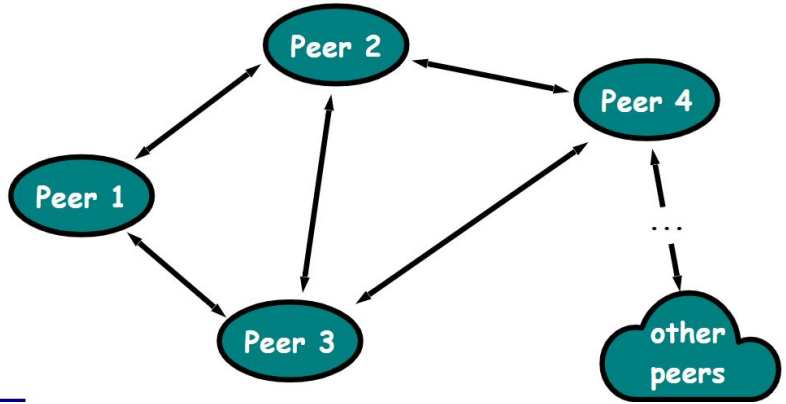
- Understandably, the diagram beside is illegible as it provides the Calls and CalledBy for the src as a whole, however it provides interesting insight as to size, clustering and interactions of some of the major and heavy components.

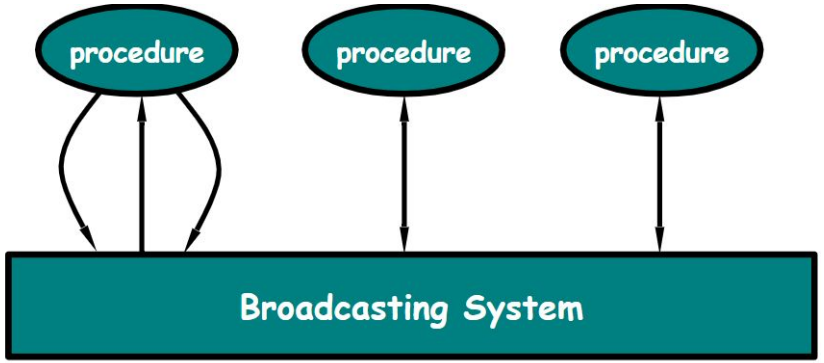
- Top Right: database – responsible for storing information.
- Middle Left: RPC – responsible for the interactions with the blockchain including coin validation, transactions and wallet.
- Middle Center: bench – responsible for. Bottom Left: wallet – responsible for much of the processing of all banking operations at the user level.
- Bottom Middle: sketch – responsible for the serialization and deserialization of data sent through the platform.
- Bottom Right: secp256k1 – responsible for the public key cryptography on the platform.



Architectural Style

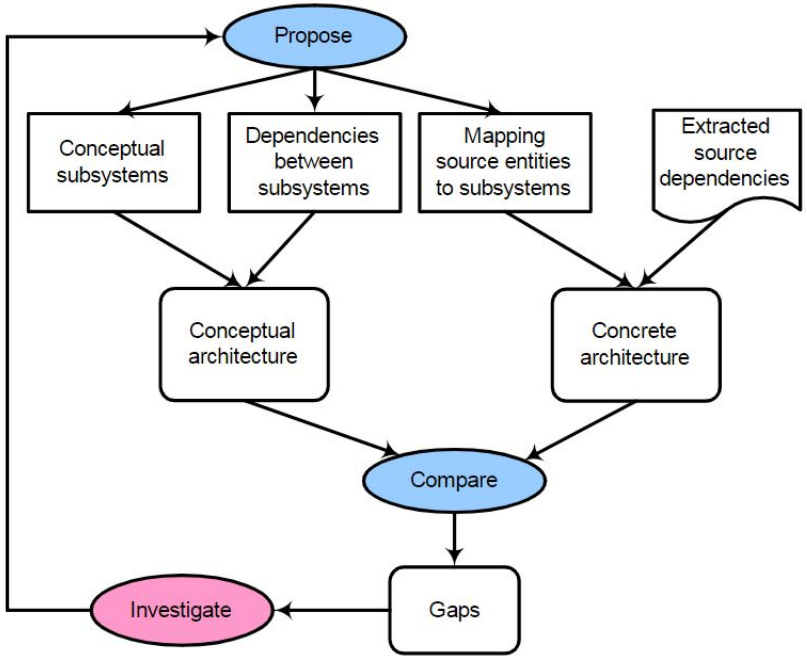
- As mentioned above, the outside layer of the Bitcoin Core software is pub-sub. The interfaces module allows the user to interact with the platform while the node module connects with the larger chain. Within the software, elements are connected to each other through an object-orient network while maintain the pub-sub style.





Reflexion Analysis: High-Level Architecture

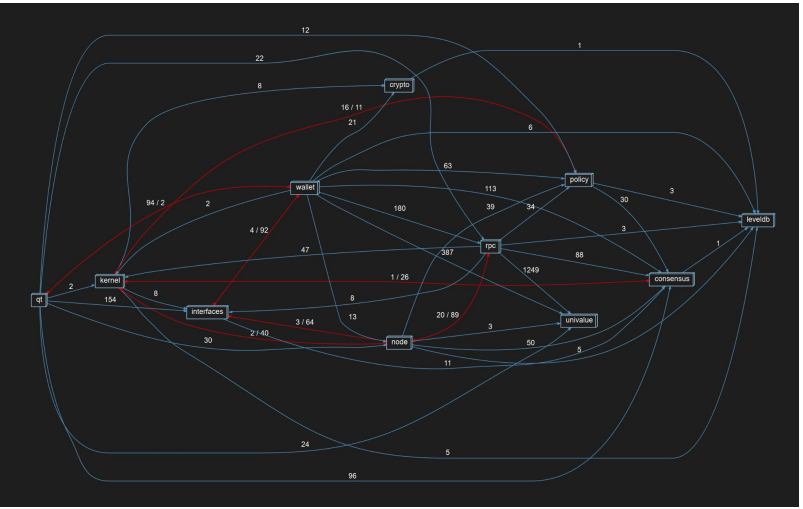
The next slides will cover the high level Reflexion Analysis, new systems, and new and unexpected dependencies



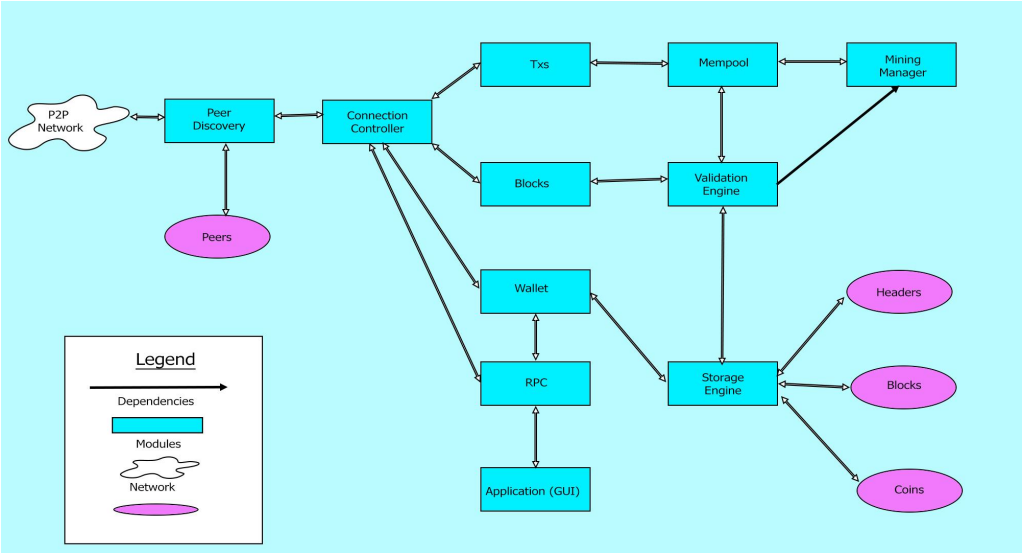
High-Level: New Subsystems

- A new concrete architecture for Bitcoin Core has been derived and several new subsystems have been identified, including Node, Interfaces, Kernel, Crypto, Policy, Univalue, and Consensus subsystems.
- Some subsystems that were included in the conceptual architecture were also found in the concrete architecture but with different names in the files, such as QT subsystem (GUI/Application), levelDB subsystem (Storage Engine), and validation Engine (Policy subsystem).
- The Node subsystem accesses the node state in the network, while the Interfaces subsystem defines interfaces between major components of the code, such as Node and Wallet, to allow for independent testing and development.
- The Kernel subsystem provides MUTEX locks and access to libraries to ensure correct resource allocation and statistics of coin operations.
- The Crypto subsystem handles ciphering keys, conversions with respect to currencies to satoshis, and various ciphers such as Symmetric ciphers, AEAD ciphers, and hashing.
- The Univalue subsystem includes JSON encoding and decoding for the abstract data type used in Bitcoin Core.
- Finally, the Consensus subsystem ensures adherence to Bitcoin rules, such as block sizes. The interactions between these subsystems and previously known subsystems will be explained in detail in the next section.

High-Level: New and Unexpected Dependencies



vs.

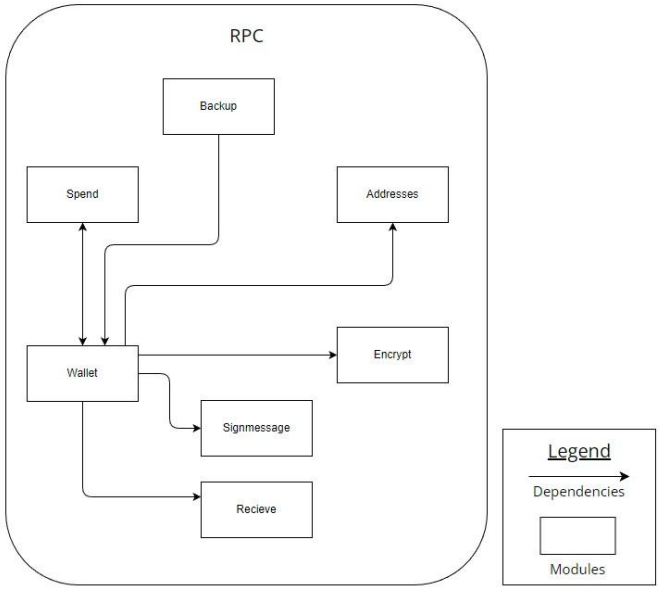


High-Level: Reflexion Analysis

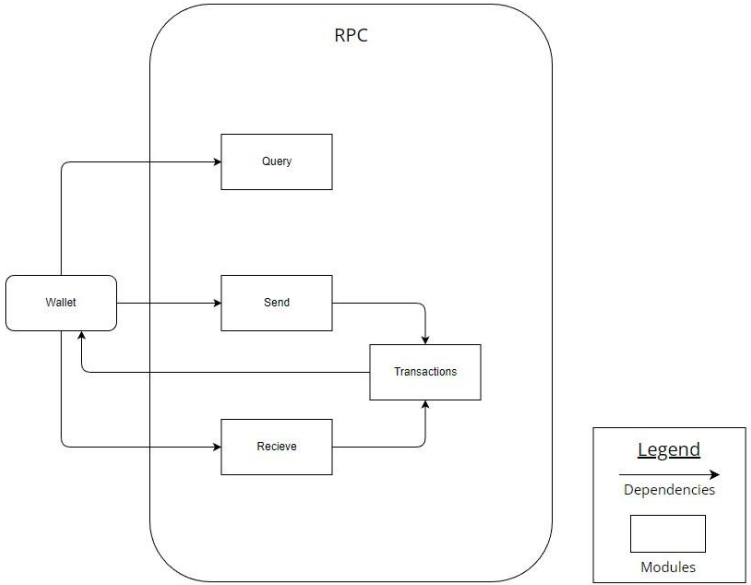
- After analyzing the concrete and conceptual architectures of Bitcoin Core, we conducted a reflection analysis on the two architectures.
- We compared the results of the two architectures in the new subsystems and new dependencies sections of the report.
- The new subsystems are Crypto, kernel, interfaces, node, univalue, and consensus subsystems.
- The consistent dependencies between the two architectures are RPC and wallet, as well as wallet with levelDB.
- The concrete architecture showed unexpected dependencies, such as the two-way dependency between QT and Wallet, which is not typical in the conceptual architecture.
- Another unexpected dependency is the one-way dependency between RPC and levelDB, not present in the conceptual architecture.
- The rest of the dependencies are new, related to the new subsystems that were not included in the conceptual architecture.
- We identified gaps between the two architectures, including absences such as P2P Network, Peer discovery subsystem, database of peers, connection manager, Txes, mempool, miner, and blocks subsystems.
- Convergences between the two views are mainly the subsystems that are in both, such as Wallet, RPC, QT, levelDB, and policy.
- There are many divergences between the two views, mainly due to new subsystems and unexpected dependencies added in the concrete architecture to accommodate the growing user base and improve security.

Reflexion Analysis: Second-Level Subsystem src/wallet/rpc

Concrete View



Conceptual View



Second-Level: Reflexion Analysis

- The RPC module turned out to be much more widely-encompassing than we originally thought.
- We did not take security or encryption of transactions into account when creating the conceptual diagram, nor did we consider that backups of keys would be an important component.
- We assumed that the outer Wallet system exists completely outside of RPC, and that RPC contains methods that expand on the Wallet system by allowing for communication with other components in the top-level.
- Instead, although the Wallet “system” does exist outside of the RPC module, the actual file/implementation for Wallet exists within the RPC module.
- This can be confusing since RPC is a subsystem of Wallet, yet implementation of it falls under the RPC module and not under the Wallet module as we originally thought.
- The entire core functionality of the Wallet system lies within its RPC subsystem.
- We assumed that the outer Wallet system would handle implementation of Addresses, rather than Addresses being its own component, and also encapsulated by RPC.
- However, the overall functionality of the RPC components we conceptualized remained largely consistent with the concrete architecture.

Second-Level: Unchanged Subsystems

- Spend: Renamed from “Send,” handles sending Bitcoin from the user's wallet to a recipient's address by creating and confirming transactions and committing them to the user's wallet.
- The function "EnsureWalletIsUnlocked" in the Encrypt component is called to ensure that the wallet is unlocked before transferring Bitcoin.
- Parameters include the address to send to, the amount to send, and optional parameters such as comment and fee rate.
- The wallet broadcasts the transaction to the Bitcoin network and its status is confirmed once verified by nodes on the network.
- Receive: Provides implementation for receiving Bitcoin by sharing the wallet-generated address with the sender.
- Once the transaction is confirmed on the Bitcoin network, the Bitcoin is added to the wallet's balance.
- Signmessage: Renamed from “Transactions,” is used to sign messages with the private key of an address.
- Signing a message with an address proves ownership of the associated Bitcoin address and creates a digital signature for the specific message.
- The signature can be verified by anyone with the public key and is used for authentication purposes.
- Signmessage is used to prove ownership of an address and to ensure that a message has not been altered during transmission.

Second-Level: New Subsystems

Wallet: The actual implementation of the wallet system. This provides the constructors to create a wallet object. In contrast, the other components of RPC mainly provide functions that are used by the wallet component. Properties of a wallet object include its name, version, database format, balance, transaction count, and key pool size.

Backup: Used for wallet key management. Used to add private or public keys to the wallet, and to store them. A public key allows a user to add an address to their Bitcoin wallet which can be watched but cannot be used to spend funds, unlike a private key. All of the keys associated with a user's wallet can be backed up to a server-side file.

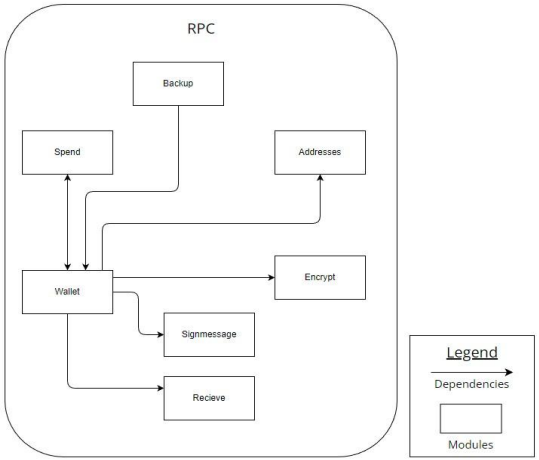
Addresses: An address is needed to identify both the sender and the recipient during a transaction. Each wallet has an associated address, similar to a bank routing number. This component generates this address. Can also function to add a multisignature address to the wallet, which specifies the number of required signatures out of a certain number of addresses.

Encrypt: The purpose of this component is to provide security for a user when access to the wallet is needed by other systems, such as when sending Bitcoin, by allowing for the wallet to be locked and encrypted. This is done by changing the passphrase, or key, used to decrypt the wallet. This key is stored in memory for a set period of time before it's changed.

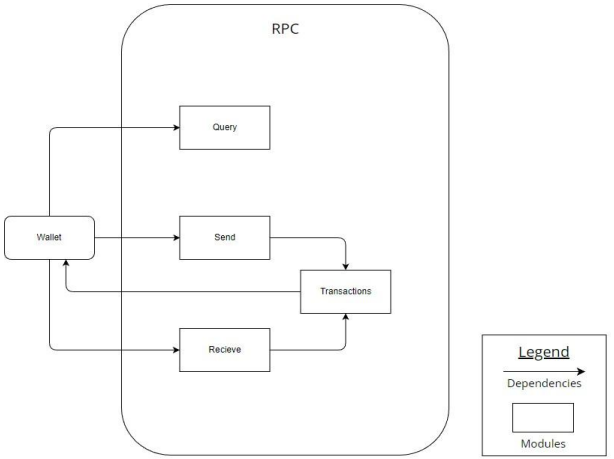
Second-Level: Unexpected Dependencies

- The Spend subsystem contained an unexpected dependency on Wallet. We did not account for two-way address authentication between sender and receiver when sending Bitcoin.

Concrete View

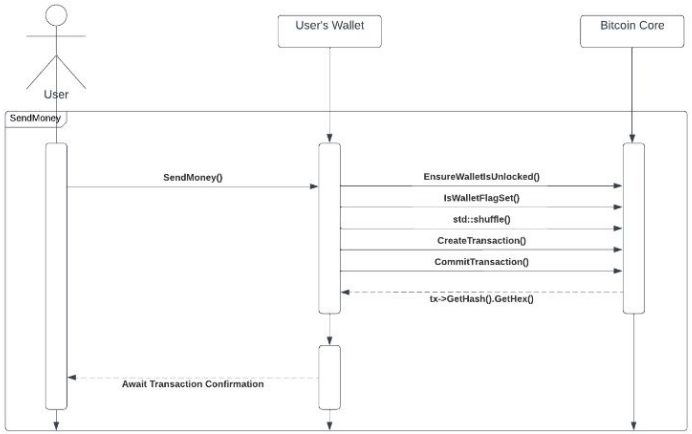


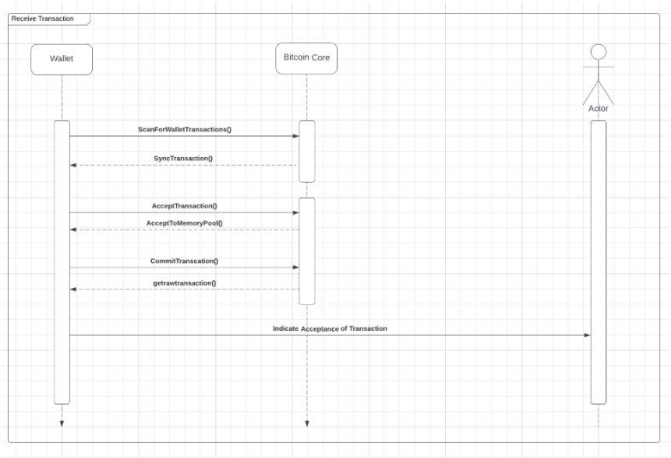
Conceptual View



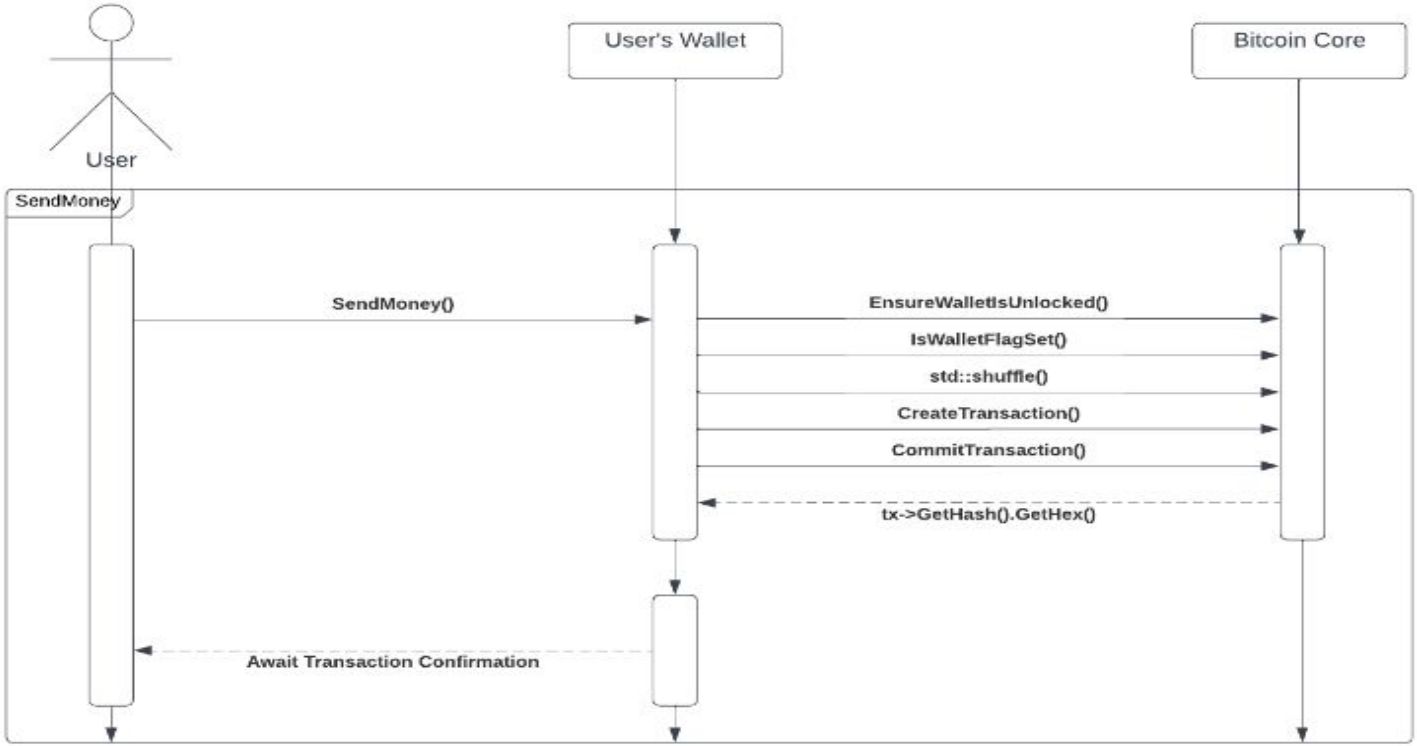
Use Cases

- There are generally two common use cases
  - 1: Sending money utilizing the wallet
  - 2: Receiving the money through the wallet





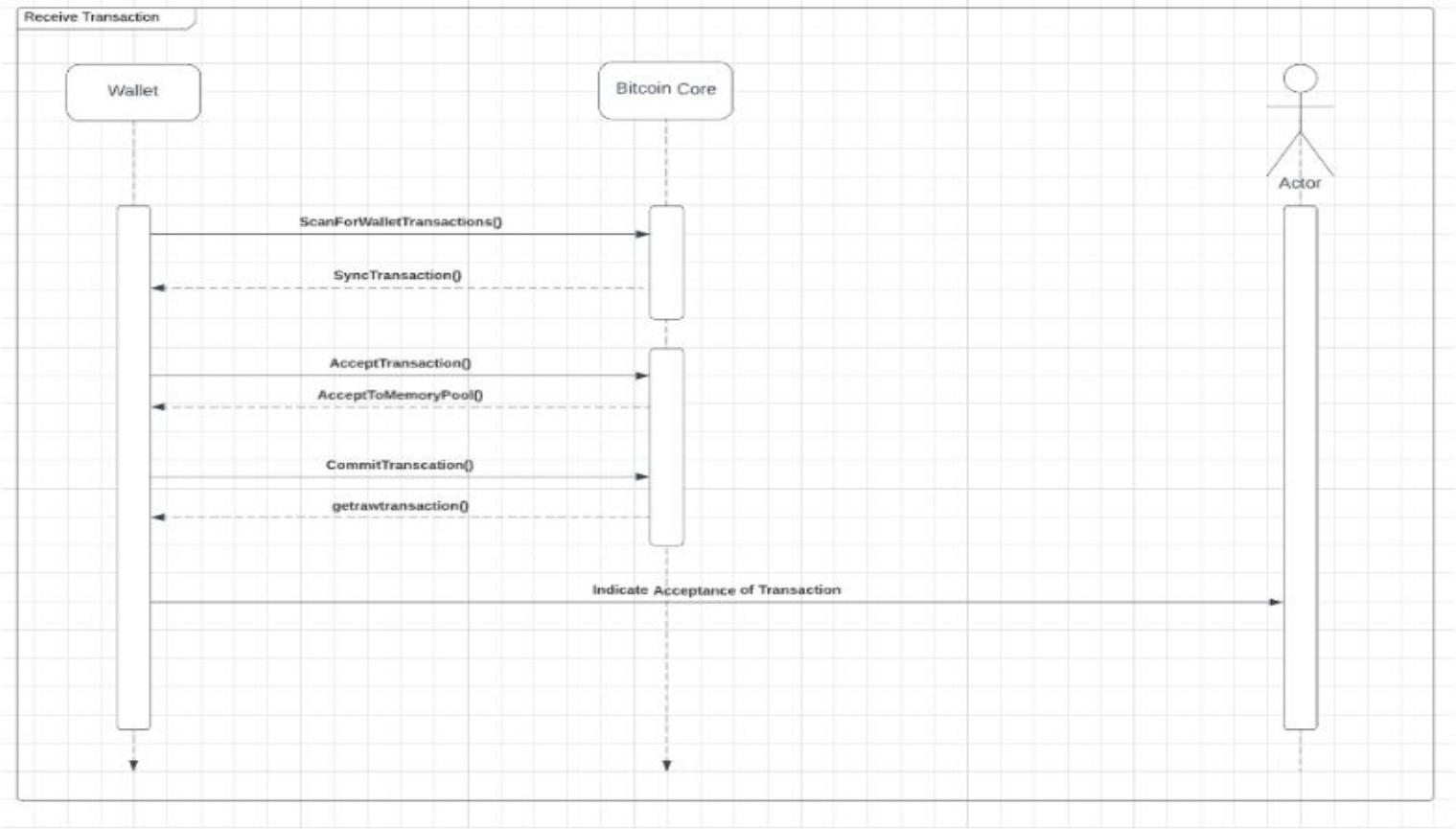
Use Case 1: Sending Money With the Wallet



Use Case 1: Walkthrough

When a user clicks "Send Money" on the wallet app, it first checks if the wallet is unlocked and has private keys enabled. Then, it shuffles the recipient list to randomize the transaction order for security. The CreateTransaction method constructs the transaction with a calculated fee, and CommitTransaction updates the wallet's balance and transaction history. The resulting transaction ID is obtained through the tx->GetHash().GetHex() method, which returns a hexadecimal string. The app then sends a JSON confirmation to the user.

Use Case 2: Receiving Money With the Wallet



Use Case 2: Walkthrough

When accepting and processing a new transaction on the Bitcoin network, the wallet first scans for transactions using the ScanForWalletTransaction() function. If a valid transaction is found, it undergoes verification with the SyncTransaction() function to prevent double-spend attempts. Once confirmed, the AcceptTransaction() function accepts the transaction, and Bitcoin Core accepts it into the memory pool using the AcceptToMemoryPool() function. The newly created transaction is then committed using the CommitTransaction() function, which updates the wallet's transaction list and balance. Users can see their updated balance when they check it.

Limitations and Lessons Learned

- Some limitations that were encountered throughout this assignment were
  - Size of the source code
  - Understanding the diagrams generated by Understand
  - Computational limits while using Understand
  - Building the sequence diagram

- Some lessons learned were
  - The importance of team work and group collaboration
  - Organizing group meetings asynchronously is an asset
  - How great a resource source code documentation can be