CISC 332/326 - Assignment 2 - Report

Bitcoin Core - Concrete Architecture Analysis

Friday, March 24th, 2023

Deluca, Seb – 20sad4@queensu.ca

Huang, Robbie – 20rh1@queensu.ca

Kim, Ethan – 20eik@queensu.ca

Leyne, Aidan – 19afl@queensu.ca

Liang, Sean – sean.liang@queensu.ca

Lomonaco, Artie – 19al63@queensu.ca

Contents

Abstract

In this report, we will present the analysis of the concrete architecture of Bitcoin Core. This open-source software is the client-side interface for the Bitcoin protocol: “an open source Bitcoin fork, styled around the principles of cheap transactions, on-chain scalability and reliable direct payment networks”. Having proposed a conceptual architecture for the system in an earlier report, this report will focus on the concrete architecture. Through an outlined derivation process, we will examine the platform in two main stages. Finally, we will look at concurrency within the system, along with presenting some of the limitation and learning lessons arising from the process.

High-Level: We will complete a reflexion analysis, comparing and contrasting the conceptual and concrete architectures, examine the subsystems comprising the architecture and present use-cases to show flow within the system.

Low-Level: Having chosen the *src/wallet/rpc* subsystem, we will complete the same process as described above. For this section however, we will also present the conceptual architecture for the subsystem, as it is not yet derived.

Introduction and Overview

The objective of this report is to analyze and discuss the concrete architecture of Bitcoin Core. The Bitcoin Core system leverages the P2P architectural style. P2P is shorthand for Peer-to-Peer, a decentralized platform wherein individuals utilizing the service interact directly with each other, rather than having to interact with each other through a server. Bitcoin Core also utilizes the Publish/Subscribe architectural style for handling transactions, among other things. Our concrete architecture is derived accordingly, also considering the newly-provided source code. The first part of the report will be regarding the derivation process– describing how the concrete architecture was visualized with Understand. Then, we will look into Bitcoin Core's concrete architecture. This will contain a review on our previous conceptual architecture, as well as a detailed description of the concrete architecture with a graph. Next, we will perform Reflexion Analysis, where we will introduce new subsystems and also the new, unexpected dependencies. Following this, we will introduce a specific second-level subsystem that is a version of RPC specific to the wallet component. Then, for the use cases, we will continue with our use cases from the previous report– checking wallet balance and sending transactions. For this section, we will update our sequence diagrams to consider the new information provided by our new understanding of the concrete architecture of Bitcoin Core.

Lastly, we will leverage our discoveries with our work to formulate a conclusion, and determine what we will work on in the future.

Derivation Process

To begin formulating the concrete architecture for Bitcoin Core, we began by downloading the provided source code and loading the repository into the program Understand. This software tool allows for the dynamic visualization of large code repositories, highlighting dependencies, calls and overall architecture. Before we began to use the tool, we reviewed our conceptual architecture, along with the feedback provided from the last assignment, in order to re-familiarize ourselves. This step made it much easier when identifying known modules within Understand. This cannot be said for all modules; as we will see later on in our reflexion analysis, many modules were hiding, missing or new modules were present.

One of the first items we noticed about the Bitcoin Core codebase was the relatively unstructured nature of files and the lack of use of hierarchal directories. Within the *src* folder, it is rare to see more than two subfolders before reaching a layer of strictly files. This did make understanding the concrete architecture and the second-level subsystem architecture much more difficult as the architecture became much more file-dependant rather than structure-dependant as we have seen in other examples. Removing lone files found in the *src* folder proved useful as the dependency graph for the high-level architecture become much easier to understand. Other folders were then hidden from the dependency-view of the directory as we found them to be much more supplemental, rather than architecture-related. In contrast, this technique was not possible with the derivation for the second-level *rpc* subsystem as there are only files for this module. Instead, filenames, method names and commit messages; we were able to distinguish the roles of each of the files. Overall, the best view of the concrete architecture for Bitcoin Core is below.

Concurrency, External Interfaces and System Evolution

As the Bitcoin Core platform operates on a pub-sub model in relation to the blockchain, each peer is able to opt-in and out of the chain at any given time and return later. Internally, as we will see later in the report, it is paramount that all functionality remains concurrent with each other. This is not only for the blockchain requirements for transaction validation but also so that users are presented with accurate data, for example the state of their wallets.

The Bitcoin Core platform has two main external interfaces, the Bitcoin blockchain and the user GUI. The main functionality of the platform allows user to transact on the network, mine new coins through transaction validation and view its state. All this information is then presented to the user, the other external interface, so that they can utilize the platform.

First released in 2014, the Bitcoin Core platform is presented by the Bitcoin Organization and developed through an open-sourced system and reducing transaction costs ten-fold. Developers from around the world collaborate on the public repository found on GitHub in order to present worldwide releases. One of the largest of these improvements took place in 2016 with the release of the *CheckSequenceVerify* fork.

Concrete Architecture: Bitcoin Core

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.

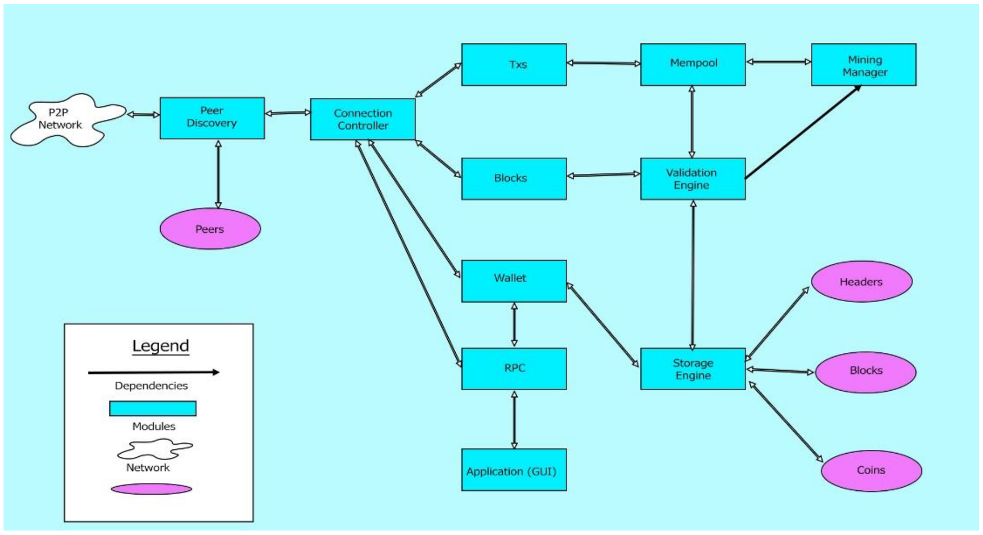


Figure: Conceptual Architecture of Bitcoin Core

Concrete Architecture

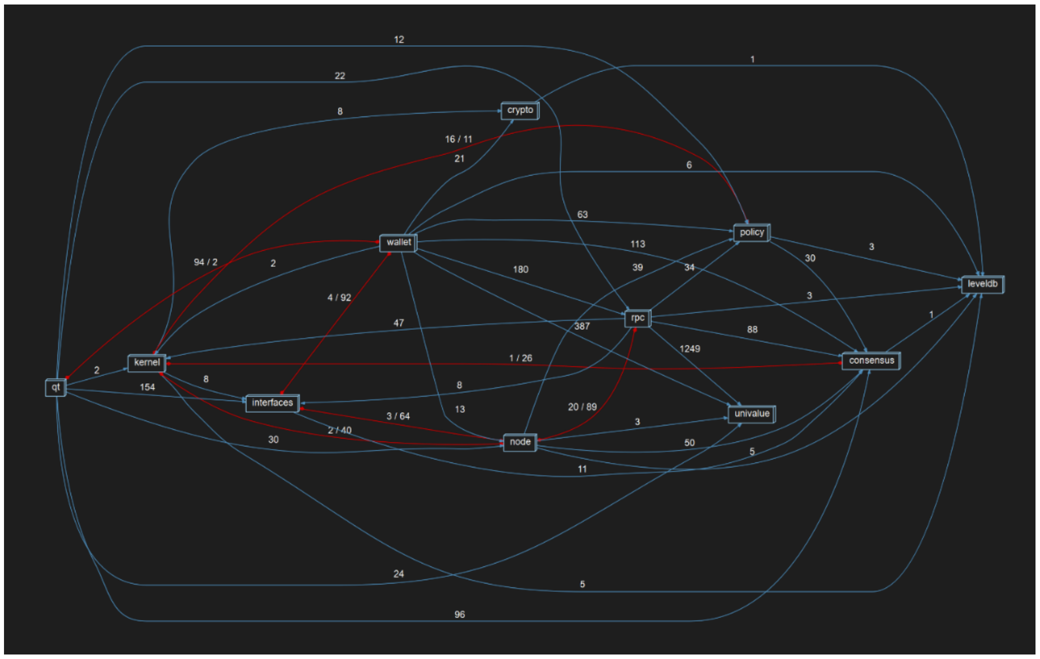


Figure: Concrete Architecture of Bitcoin Core

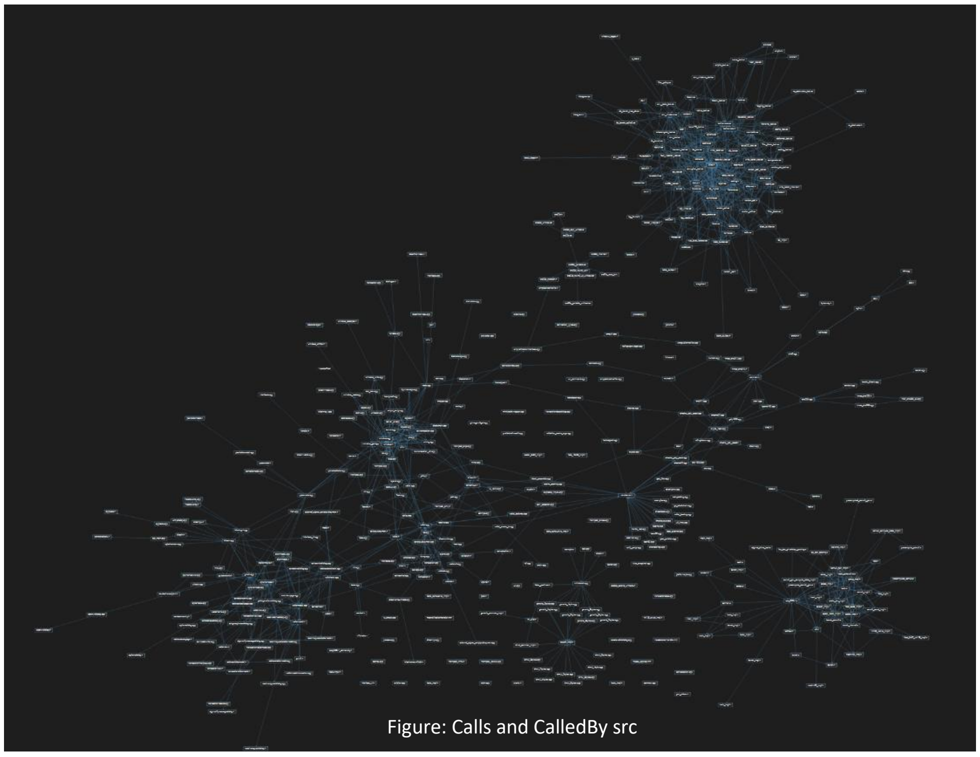


Figure: Calls and CalledBy src

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. *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.

Architecture 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

New Subsystems

In the new concrete architecture that we derived, many new subsystems have been found in bitcoin core. It might seem that they are all completely new, but some of them are the same subsystem that we included in our conceptual architecture but they are named differently in the files. The new subsystems that were found in the concrete architecture are the Node, Interfaces, Kernel, Crypto, Policy, Univalue, and Consensus subsystems.

The subsystems that were renamed are QT, which is the GUI/Application subsystem in the conceptual architecture. The levelDB subsystem is the Storage Engine from the conceptual architecture. The validation Engine in our conceptual architecture is the policy subsystem in the concrete architecture.

Firstly, the *node* Subsystem is the directory that accesses the node state in the network. The code in this subsystem is isolated from Wallet and QT so that they do not interfere with Node operation, and also to allow those two to run separately from Node. Next, the Interfaces subsystem, where many of the interfaces in the system are defined. Some of the interfaces defined are Node and the Wallet. The interfaces that are defined are boundaries between major components of the bitcoin code, so that they are all able to run in different processes, tested, developed, and understood independently. The Kernel subsystem provides MUTEX locks and access to a few libraries to ensure that resources are allocated correctly when the program is running. It also gets the statistics of the operations that have to do with coins, such as getting the amount of coins that were lost due to unclaimed miner rewards. The final thing that is included in the kernel is the mempool which stores data of the transactions. Next, we will be talking about the crypto subsystem. The crypto subsystem is used for ciphering keys by calling APIs. Some of the ciphers include Symmetric ciphers, AEAD ciphers, message digest/including keyed message digest, random number generation, user space interface, and hashing. This subsystem also handles conversions with respect to currencies to satoshis. Next, the Univalue directory with JSON encoding and decoding. The UniValue is an abstract data type that is used in the system of Bitcoin Core. Finally, the consensus subsystem ensures that the bitcoin rules are being followed, such as the block sizes and other rules. In the next section the interaction between these subsystems, and the known ones will be explained in detail.

New and Unexpected Dependencies

After a thoughtful comparison of both the conceptual and the concrete architecture diagrams that we derived, we have found that there are a great number of differences between them. Some dependencies are similar, but the majority are different or new, especially since there are many new subsystems in the concrete architecture. The number of dependencies and the type of dependency will be outlined for each subsystem, as well as the comparisons to the conceptual architecture.

Now to explain the dependencies of conceptual vs concrete architecture. Firstly, the dependencies of RPC and Wallet. The dependency between the RPC and the Wallet is only a one way dependency, with 180 dependencies, where the Wallet depends on RPC, unlike in the conceptual architecture where it is a two way dependency. In the concrete architecture there is also a one way dependency on levelDB, while in the conceptual it was a two way dependent. In the concrete architecture there is now no dependency between the RPC and the QT, but it is a two way (94/2, QT has 94 dependencies on *wallet*, and 2 the other way dependency between the wallet and the QT, this is one of the unexpected dependencies. The other two way dependency that Wallet has is with the interfaces subsystem with (Wallet -92-> interfaces, and interfaces -2-> Wallet). The new dependencies found that are one way from the Wallet subsystem are 13 dependencies of Wallet on Node, 63 dependencies on the policy subsystem, 21 dependencies on the crypto subsystem, and 387 dependencies on UniValue subsystem. Apart from those, we are missing a dependency from the Wallet to the levelDB (Storage Engine) and the connection manager is not present in the concrete model. Next the dependencies that RPC has on other subsystems will be explained. RPC has a two way dependency with Node (RPC -89-> Node, and Node-20-> RPC). The other dependencies that RPC has on other subsystems are 47 on Kernel, 3 on levelDB which is a unexpected dependency sine normally it is the Wallet that has a dependency with it, 88 dependencies on consensus, 34 on policy subsystem, and finally 1249 dependencies on univalue. Next, Node has 3 two way dependencies with interfaces (Node -64-> interfaces, and interfaces -3-> Node), with Kernel (Node -40-> kernel, and kernel -2-> Node), and with RPC which has already been explained. Node also has three one way dependencies, 5 with levelDB, 387 on policy, and 50 on consensus. Next, univalue does not have any dependencies on other subsystems. Next, Kernel has three two way dependencies, with Node which has already be explained, a two way dependency with the consensus subsystem (Consensus -26-> kernel, and kernel -1-> Consensus), and a two way dependency with policy (policy -11-> kernel, and kernel -16-> policy). Kernel also has three one way dependencies: 8 on crypto, 5 on the levelDB, and 8 on interfaces subsystems. Next, the QT or GUI/Application subsystem has a two way dependency with Wallet which has already be pointed out, and it has one way dependencies: 12 on policy, 22 on RPC, 2 on kernel, 154 on interfaces, 30 on node, 24 on univalue, and finally 96 dependencies on levelDB. The next subsystems only have one one-way dependency on other subsystems. Interfaces have 11 dependencies on consensus, crypto has 1 dependency on levelDB, policy has 3 dependencies on levelDB, and consensus has 1 dependency on levelDB.

To summarize, the dependencies that were consistent between the two architectures are RPC and wallet (but is one way instead) and wallet with levelDB (also one way instead). One of the unexpected dependencies seen in the concrete architecture is the two way dependency between QT and Wallet, where normally it is RPC that has this dependency instead of wallet in our conceptual architecture. Another one is that RPC has a one way dependency with levelDB which is also not in our conceptual architecture. The rest of the dependencies are new since they are with the new subsystems that were not included.

Second-Level Subsystem: *scr/wallet/rpc*

Conceptual Architecture

RPC (Remote Procedure Call) is an interface within the wallet top-level subsystem that provides a way for the Wallet subsystem to communicate and interact with other components in Bitcoin Core. External applications, such as payment processors and merchants, can also use RPC commands to communicate with the wallet subsystem to perform functions such as creating and signing transactions, and querying wallet information. The overall architectural style for RPC is object-oriented.

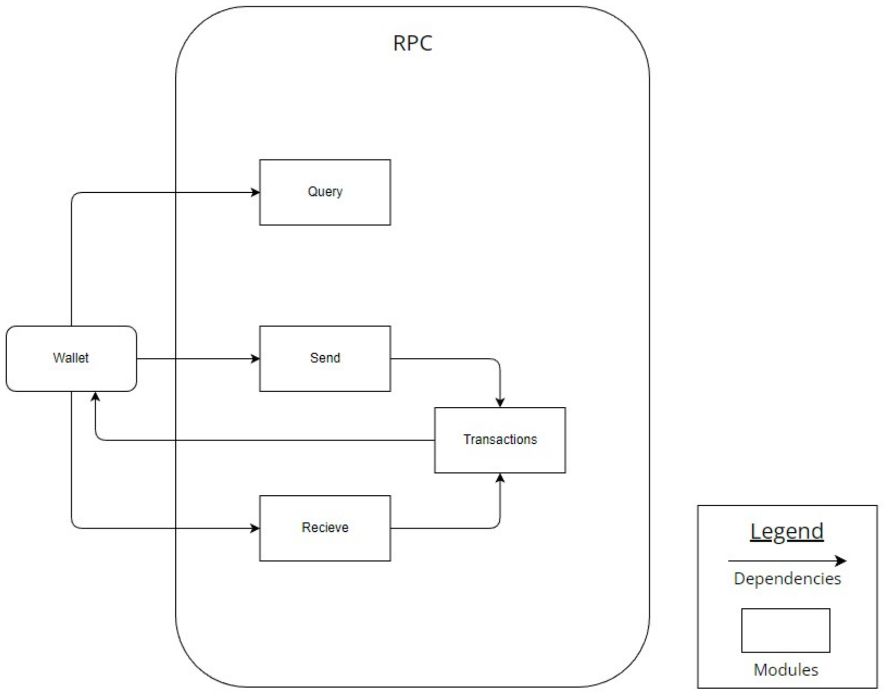


Figure: Conceptual view of RPC

Concrete Architecture

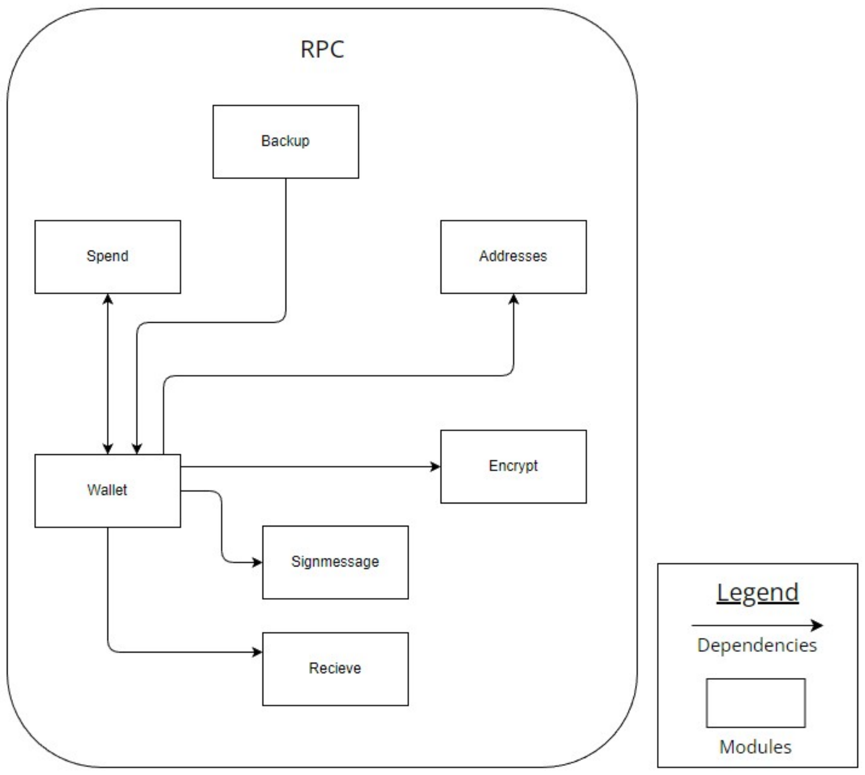


Figure: Concrete view of RPC

Subsystems and Interactions

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.

Unchanged Subsystems

Spend: Renamed from “Send.” Handles the sending of Bitcoin from the user’s wallet to a recipient’s given address. Creates transactions and confirms the status of them (e.g. if a transaction is complete or not), and commits transactions to the user’s wallet. Ensures that the wallet is unlocked first before transferring Bitcoin by calling the function “EnsureWalletIsUnlocked'' located in the Encrypt component. Parameters include the address to send to, the amount to send, and optional parameters such as a comment and fee rate. The wallet then broadcasts the transaction to the Bitcoin network and its verified by the nodes on the network. Once the transaction is verified, it's added to the blockchain and its status is set to "confirmed."

Receive: Provides implementation for receiving Bitcoin. The address generated by the wallet will be shared with the sender. Once the transaction is broadcasted and confirmed on the Bitcoin network, the Bitcoin will be added to your wallet's balance.

Signmessage: Renamed from “Transactions.” This component is used to sign messages with the private key of an address. Signing a message with an address means that you are proving that you are the owner of a particular Bitcoin address by using the private key associated with that address to create a digital signature for a specific message. The signature can then be verified by anyone who has the public key associated with the address, and is proof that the message has not been altered during transmission. It is also often used to prove ownership of an address for authentication purposes.

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.

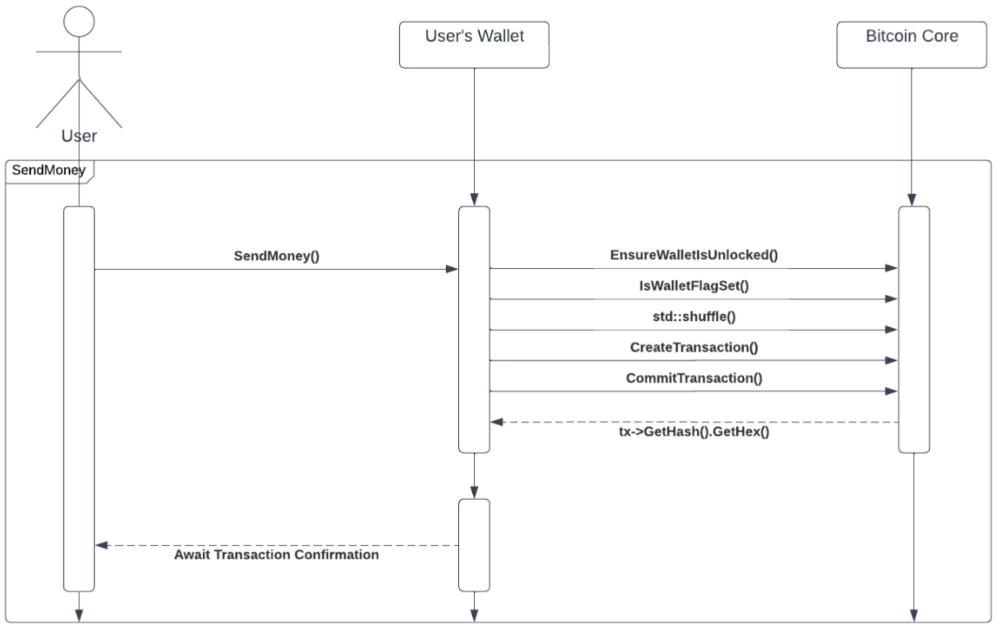
Unexpected Dependencies

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.

Use Cases

Use Case 1: Sending Money with Wallet

The wallet component of Bitcoin Core contains a file named *spend.cpp*. *spend.cpp*, which will be hereafter called *spend*, is responsible for creating, committing, and sending transactions from a Bitcoin wallet, as well as performing updates to a wallet's transactional history and updating its balance. In this use case diagram, we will consider the circumstances in which a user will attempt to send money from their Bitcoin wallet to another user. We will depict all the functions that Spend performs to complete such a task.



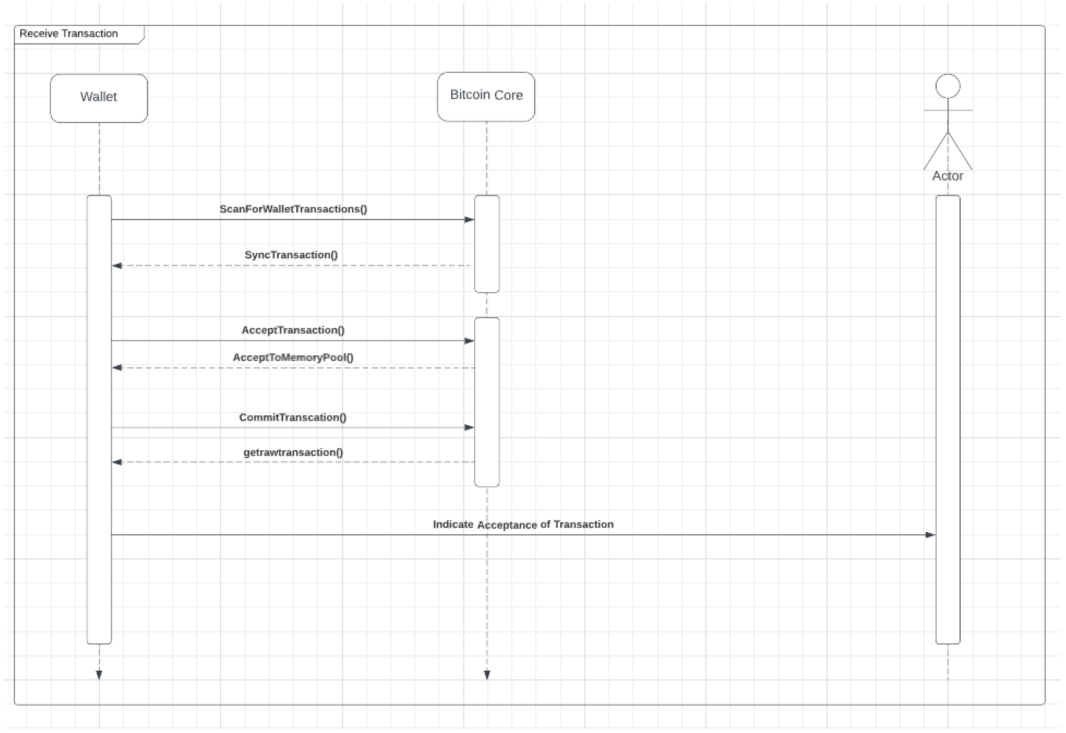
Walkthrough

1. The user clicks the 'Send Money' button in the UI for the wallet.
2. The wallet application, User's Wallet, communicates with Bitcoin Core to ensure that the wallet is unlocked using *EnsureWalletIsUnlocked*.
3. Then, User's Wallet runs the *IsWalletFlagSet* method to check if a specific flag is set for the wallet to check if it has private keys enabled or not.
4. Then, User's Wallet runs the *std::shuffle* method to randomly shuffle the recipient list into a new order to ensure that the order that outputs for transactions are created is randomized. This is done in an effort to make it more difficult for an attacker to identify a transaction.
5. Then, User's Wallet runs the *CreateTransaction* method, which constructs a transaction that meets the given specifications. This will return a *CAmountOrError* object which contains the constructed transaction and a calculated transaction fee. \
6. Afterwards, the method *CommitTransaction* is used to add the newly created transaction to the wallet's list of transactions, as well as update the wallet's state to reflect the new transaction (i.e updating balance).

7. Bitcoin Core then runs the *tx->GetHash().GetHex()* method. This is broken up into two parts.
    a. *tx->GetHash()* returns the hash of the transaction object.
    b. *GetHex()* converts that hash into a hexadecimal string.

   The resulting hexadecimal string is the transaction id, called *txid*, which can be used to track the transaction on the Bitcoin network.
8. Finally, the wallet will return a JSON response to the user indicating the confirmation of the transaction

Use Case 2: Receiving Money with Wallet



Walkthrough

1. The wallet will query the Bitcoin Network for transactions using *ScanForWalletTransaction()* function to find a transaction to be accepted
2. Once it has found a transaction it will use the *SyncTranscation()* function to process the new transaction that has been received from the network. It will also ensure that the transaction is valid and not a double-spend attempt
3. Once it passes through the *SyncTranscation()* function, it will accept the transaction using the *AcceptTranscation()* function.
4. Once the transaction is accepted, Bitcoin Core will then accept it into the *memorypool* using the *AcceptToMemoryPool()* function
5. The wallet then commits the transaction using the *CommitTranscation()* function, which adds the newly created transaction to the wallet's list of transactions, and update the wallet's balance
6. Now when the user checks the balance it will reflect the updated transaction

Limitations and Lessons Learned

Throughout our experience using the Understand software, we have gained valuable insights into the importance of patience and perseverance in source code analysis. The size of the source code presented a challenge, requiring us to invest a significant amount of time to load and interact with it effectively. Additionally, we had to engage thoroughly in analysing and carefully reading the documentation to comprehend the role and interactions of each component, proving to be a time-consuming task. Furthermore, we faced the challenge of deciphering messy diagrams, further complicating our understanding of the inter-component interactions. Nevertheless, we learned that investing time and effort in understanding the software architecture is crucial for the success of any software development project.

The importance of teamwork was also highlighted during our project. Managing our time effectively and aligning our schedules proved to be a challenge. The busy schedules of all team members necessitated the holding of asynchronous meetings to move the project forward. Moreover, communicating with the TA was challenging, requiring persistent email correspondence to organize team-wide convenient meetings. Our experience taught us that effective teamwork is vital in achieving project goals and delivering quality results.

Building accurate sequence diagrams highlighted the importance of the course material. In creating the sequence diagram, we had to employ strong deductive reasoning, based on the insights derived from the Understand program. We learned that effective utilization of the documentation is crucial to understanding the components and linking them in a communicable manner within the report. Our experience taught us that investing time and effort in studying the materials provided is crucial in creating a robust software architecture.

Conclusion

After carefully analyzing and discussing the concrete architecture of Bitcoin Core, it is clear that the system leverages a decentralized platform with P2P architectural style. The Publish/Subscribe architectural style is also utilized for handling transactions, among other things. Through the derivation process, we were able to visualize the concrete architecture with Understand and provide a detailed description with compelling sequence diagrams and subsystem visualizations. We also performed a reflexion analysis, introducing new subsystems and unexpected dependencies, including a specific second-level subsystem that is a version of RPC specific to the wallet component.

Data Dictionary, Naming Conventions and Abbreviations

RPC: Remote Procedure Call is a protocol for requesting services located in another machine without requiring the details of the network that machine is located on.

P2P: Peer-to-Peer commonly refers to the decentralized ledger protocol used in blockchain technology; the name derives the connection of individual peers, lacking a central authority.

Subsystem: An individual computational environment where resources and workflow are coordinated.

Payment processors: Middlemen in the transaction between payee and merchants on the Bitcoin network.

Merchants: Business that sells products or services along the network.

Querying: Process of requesting information along the bitcoin network.

Consensus verification: Consensus verification is the process by which Bitcoin miners compete to solve a mathematical puzzle, ensuring the validity of new transactions added to the blockchain.

Decentralized: Decentralization is the process of distributing power and control away from a central authority, empowering a network of participants to collectively make decisions and govern the system.

Wallet: A stash for cryptocurrencies and crypto derived assets.

Block hash: The block hash is a unique identifier that is generated for each block in the Bitcoin blockchain, using a cryptographic algorithm to ensure the block's integrity and immutability.

Block height: Block height is the number of blocks that have been added to the Bitcoin blockchain before a particular block, indicating its position within the blockchain.

Fee rate: Fee rate is the amount of fees paid per unit of transaction size in a Bitcoin transaction, which determines the priority and confirmation time of the transaction in the blockchain

Pub/Sub: A pub/sub architecture is a messaging pattern where publishers send messages to topics or channels, and subscribers receive those messages, enabling asynchronous communication and decoupling of components in distributed systems.

Multisignature: Multisignature is a digital signature scheme that requires multiple signatures to authorize a transaction, providing enhanced security and accountability for digital applications such as cryptocurrency wallets.

References

Bitcoin Core integration/staging tree. (2023, February 20). GitHub. https://github.com/bitcoin/bitcoin/blob/master/doc/JSON-RPC-interface.md

Developer Guides — Bitcoin. (n.d.). Developer.bitcoin.org. https://developer.bitcoin.org/devguide/index.html

Transactions - Mastering Bitcoin [Book]. (n.d.). www.oreilly.com. Retrieved February 20, 2023, from https://www.oreilly.com/library/view/mastering-bitcoin/9781491902639/ch05.html#:~:text=The%20wallet%20calculates%20the%20user

A Blockchain Glossary for Beginners. (n.d.). ConsenSys. https://consensys.net/knowledge-base/a-blockchain-glossary-for-beginners/

Bitcoin Core - Bitcoin Wiki. (n.d.). En.bitcoin.it. Retrieved March 25, 2023, from https://en.bitcoin.it/wiki/Bitcoin_Core#Version_history

Bitcoin Core integration/staging tree. (2023, March 25). GitHub. https://github.com/bitcoin/bitcoin/blob/master/doc/JSON-RPC-interface.md#:~:text=The%20RPC%20interface%20allows%20other

Kernel Crypto API Architecture — The Linux Kernel documentation. (n.d.). Www.kernel.org. Retrieved March 25, 2023, from https://www.kernel.org/doc/html/latest/crypto/architecture.html