Bitcoin Core - Concrete Architecture

Date: March 11th, 2023

Prepared by:

Daniella Ruisendaal (Team Lead) – 17dar8@queensu.ca

Alina Padoun (Presenter) – 18ap1@queensu.ca

Adam Ciszek (Presenter) – 19ac80@queensu.ca

Aidan Wolfson – 18aaw1@queensu.ca

Camila Izquierdo – 18cni@queensu.ca

Tanner Big Canoe - 19tjb8@queensu.ca

Table of Contents

Abstract

This paper will be discussing the architecture of Bitcoin Core. With the conceptual architecture and the high-level concrete architecture discussed as a whole, as well as the conceptual and concrete architecture for one of its second level subsystems – the bitcoin wallet. The software "Understand" was used to get a deeper look at these subsystem architectures and glean more information as to how they work and interact with each other, including any dependencies between them. The concurrency of Bitcoin Core is also an important aspect to its architecture, with this paper giving a more in depth look at its processes than previously discussed.

Introduction

A conceptual architecture was proposed for the Bitcoin Core system and its subsystems in Assignment 1, which was built around providing users with the chance to take part in the bitcoin network as a full node. This allowed users to send and receive payments, verifying and validating all transactions, and engage in mining. The blockchain is a public ledger that timestamps bitcoin transactions that are then publicly stored on each user's end. It interacts with the transactions and mining modules to verify and add blocks to the ledger. Some other modules that need to interact with the transactions module are the payment subsystem and the wallet subsystem, which creates and stores public and private keys to be used during transactions. The mining subsystems allow users to solve hash puzzles in order to make new blocks and also validate those blocks to be allowed into the blockchain. Overall, the Peer-to-Peer network subsystem is the most essential part of the architecture, as it allows for the actual decentralized network. It works with nearly every subsystem to fulfill the functions of Bitcoin Core.

Since the conceptual architecture has been successfully identified, now the concrete architecture of Bitcoin Core must be derived to find any deviations. The first step to building the concrete architecture was studying the provided source code and mapping out all the dependencies, from which previously missing subsystems were identified. The concrete architecture does in fact implement a Peer-to-Peer subsystem as previously thought, so all that needed to be done was update the new submodules and their corresponding dependencies. These new submodules included an Application Programming Interface (API), consensus algorithm, and some cryptographic libraries. Some additional dependencies were present between the validation and consensus subsystems, and the transaction and crypto libraries.

Next, the wallet subsystem was more closely examined to further highlight the differences and similarities between its conceptual and concrete architecture. The wallet subsystem creates and uses public and private keys to send, receive, and spend available bitcoin. However, in the conceptual architecture, it was believed that the bitcoin value existed directly within the wallet, but through examining the concrete architecture, this was proven to be false. Instead of holding the bitcoin directly in the wallet, only the private key lies there. From there, the key is used to verify the ownership and value of the currency to send or spend on the blockchain. In

the concrete architecture, this difference is shown by dependencies on the API, storage, and validation subsystems. In addition, information is shared with the mining, peer discovery, and consensus algorithm subsystems amongst others. Lastly, there were more unexpected dependencies found between the bench and wallet subsystems, and the qt and wallet subsystems.

Another important concept to be analyzed in the concrete architecture is concurrency. As expected through the conceptual architecture, Bitcoin Core is a multi-threaded software that uses threads, mutexes, and global locks to achieve concurrency while ensuring security between shared data structures. There are many valuable threads, but one of the most important is CCheckQueue, which handles the threads for parallel script validation for transactions in blocks. In the code, a master thread pushes verifications onto a queue and these entries are then processed by worker threads, where a mutex is used to keep one concurrent CCheckQueueControl active. Thread management is fairly straight forward, as the system completes a clean exit using Shutdown(). However, it is important to note that deadlocks could be a possible issue, more specifically inconsistent locking in which the same functions are run, but the thread locks them in a different order. To solve this potential problem, there exist debugging functions such as DDEBUG_LOCKORDER that can used to document the location and length of each lock contention.

The use cases and their corresponding diagrams were carried over from the conceptual architecture portion of the report. This includes a scenario in which User A wants to send bitcoin to User B, another where a user would like to check their account details or transaction history, and lastly the case in which a user would like to begin or finish mining from their device. Although the use cases were kept the same, the new unforeseen dependencies were added as well as the relevant function names as objects. For example, when sending bitcoin, it was previously thought that the user held their bitcoin balance directly in their wallet, however in the concrete architecture it was identified that the wallet only holds the private and public keys that are used to validate the amounts using the mining subsystem. These subsystem and dependency updates also carry through to the other use cases.

Some external interfaces that are present in the Bitcoin Core system include secondary wallets for users to store their bitcoin in external systems/wallets, and the Graphical User Interface (GUI) that must transmit information to the system. Both connections can transmit and validate transactions and connect to the blockchain.

Overall, the team became aware of the differences in subsystems and dependencies between the conceptual and concrete architecture of Bitcoin Core. This also included complexities about the wallet subsystem in depth and allowed the team to build a more complete, full picture of the architecture of Bitcoin Core.

Architecture

Conceptual Architecture

The conceptual architecture proposed for the Bitcoin Core system in Assignment 1 was built around providing users the chance to take part in the bitcoin network as a full node – allowing them to send and receive bitcoin payments, verifying and validating all transactions, and to engage in mining (adding new blocks to the blockchain). The following subsystems and their interactions were outlined in the conceptual architecture.

The blockchain is a public ledger in which bitcoin transactions are timestamped and publicly stored on each user's end. This subsystem must interact with the transaction module to verify transactions. The nodes depicted within the operating modes subsystem also implement the blockchain to validate and verify the blockchain as a client. The blockchain also interacts with the mining subsystem to add new blocks, and the P2P network subsystem to allow each full node to store a full copy of the blockchain for verifying blocks and ensuring its stability.

The P2P network subsystem is an essential part of the architecture that provides the decentralized network itself. It is responsible for validating all transactions, downloading the blockchain when new full nodes are added, and broadcasting miners' discoveries. It interacts with many other subsystems, but primarily with the blockchain, transactions, mining, and wallet. The blockchain and transactions subsystems interact with the P2P network because the blockchain is fully stored on the network, and all transactions must be verified by the P2P network before being added to the blockchain. The mining subsystem depends on the network for broadcasting new discoveries to the blockchain, and the wallet subsystem must store keys that maintain ownership of currency on the blockchain.

The bitcoin wallet subsystem creates public and private keys to be used for receiving and spending money respectively. The subsystem interacts with the blockchain because unlike traditional wallets, the currency lives on the blockchain. Wallets use private keys to verify ownership over certain blocks. The wallet subsystem also must interact with the transaction subsystem to get information from the blockchain and broadcast new transactions, and the payment processing module to manage sent and received funds.

The transactions subsystem is responsible for the transfer and validation of funds. To make this function, this subsystem interacts with the blockchain, P2P network, wallets, and payment processing. The blockchain allows for transactions to be public, transparent, and secure, while the P2P allows the different nodes in the network to verify and validate each transaction. The wallet stores the public and private keys used in transferring blockchain funds (sending and receiving), while the payment processing module processes the transactions.

The mining subsystem allows users to solve hash puzzles to construct new blocks and add them to the blockchain. The mining software must interact with the blockchain and the P2P network to validate and add the blocks to the blockchain.

The payments processing subsystem handles the transfer of funds from the payer's wallet to the payee's wallet. It must interact with the transactions module to acquire the payment address, and the P2P network and blockchain for recording the transactions and validating the payment.

The operating mode subsystem manages the level of security for nodes on the blockchain. It interacts primarily with the blockchain and P2P network subsystems and helps maintain the integrity of the bitcoin network.

High Level Concrete Architecture

Derivation Process

After studying the provided source code for the Bitcoin Core software, we were able to verify conjectures that we proposed in our original conceptual architecture, as well as the presence of new subsystems that we hadn't previously mentioned. With the help of the Understand software, we were able to break down the Bitcoin Core software into its constituent parts and map the relationships between them. This helped us visualize the dependencies. As seen in Figure 1 below, an overall concrete architecture is presented, as well as the dependencies we have referred to.

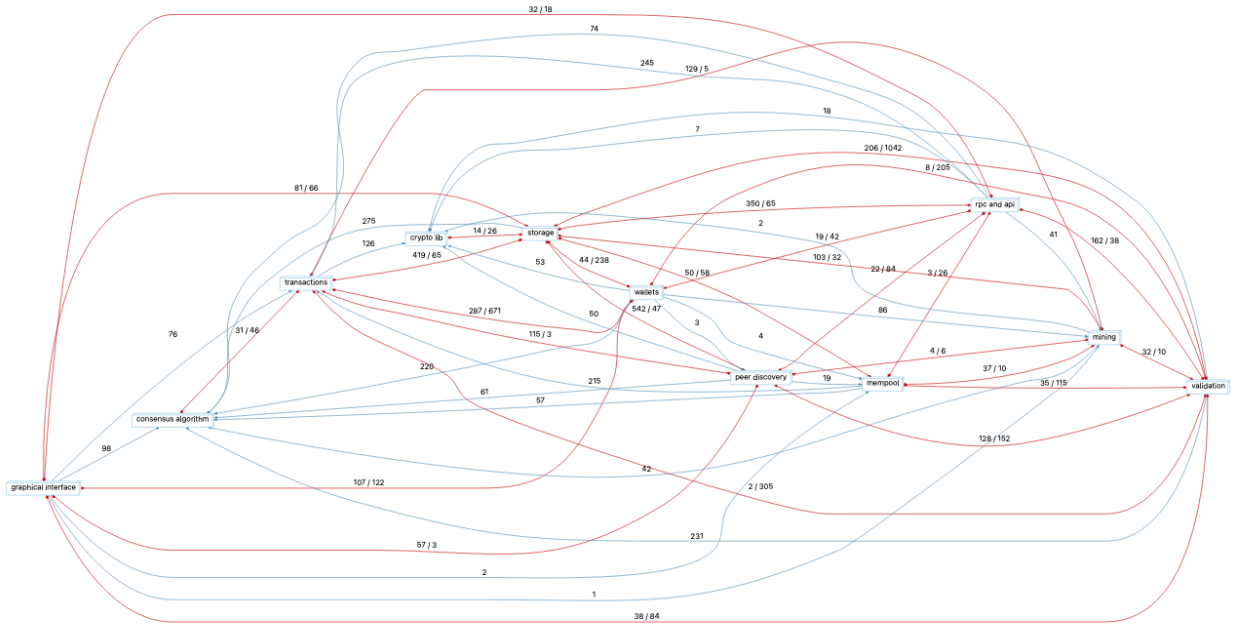

Figure 1: Dependency Diagram of Bitcoin Core as shown in Understand

The resulting concrete architecture that we derived from our analysis provides a comprehensive overview of the Bitcoin Core software, which implements a Peer-to-Peer structure as proposed in our conceptual architecture. The architecture shows the interactions and dependencies between the various components proposed in our conceptual architecture,

such as wallets, validation, peer discovery, but also includes new propositions like the API module (included in RPC), the consensus algorithm, and crypto libraries.

Reflection Analysis on Top-Level:

After reviewing the conceptual architecture and the concrete architecture, it is clear some unexpected dependencies are present, as well as new subsystems. Below is an outline of new modules that are present within the Bitcoin Core system, as well as dependencies that weren't shown in our conceptual architecture. These dependencies are discussed in some more detail on why they are necessary.

New Modules:

- Application Programming Interface (API)
- Consensus Algorithm
- Cryptographic Libraries

Unexpected Dependencies:

- Wallet → RPC
    - *Who*: Kallewoof
    - *Which*: JSONRPCRequest
    - *When*: July 4, 2019
    - *Why*: gives user access to perform different actions on their wallet, such as creating new addresses, sending and receiving funds, and managing transactions. The wallet must interact with the RPC interface to perform these user operations
- Validation → Consensus
    - *Who*: jtimon
    - *Which*: validation.h
    - *When*: May 15, 2015
    - *Why:* This is quite clear, however was not included in our dependency chart earlier. The validation.h file contains information and rules which define whether a block is valid or not, which the validation module requires. For example, the CheckBlockHeader() function in the validation module is defined in validation.h
- Transactions → Crypto Library
    - *Who*: jtimon
    - *Which*: transactions/script/interpreter.cpp depends on hash.h
    - *When*: September 8, 2014
    - *Why:* the script file depends on the hash.h file in the crypto library because this calculates cryptographic hashes of transaction data in order to validate transactions. When the script file performs different operations, the hash.h file is dependent to hash transaction data, verify digital signatures, and check validity of keys.

Conceptual and Concrete architecture of one 2nd level subsystem

What is the Wallet subsystem?

From the user's perspective the wallet is an interface that they can use to do the following: manage keys and addresses, access the user's money, track their balance, and create and sign transactions.

From the programmer's perspective at a high level the conceptual idea of the wallet is a data structure that is used to store and manage the users' keys. This was a misconception as the wallet itself does not hold any bitcoin, only keys. The coins themselves are not in the wallet but recorded in the blockchain on the bitcoin Peer-to-Peer network. The user is actually signing transactions with the keys that exist in their wallet.

Subsystem Conceptual Architecture – Wallet

Wallets are what create public keys (used to receive), and in order to spend, the corresponding private keys must be used. The difference between a traditional wallet and a crypto wallet is that the currency lives on the blockchain, rather than in your actual wallet. It is the private keys that verify ownership of this currency. This shows the significance of the P2P network, which allows users to get information from the blockchain and broadcast new transactions. This means the wallet also interacts with the transactions module. The payment processing module also manages the processing of sent and received funds, hence playing a significant role in the interaction between the wallet, other modules, and other nodes.

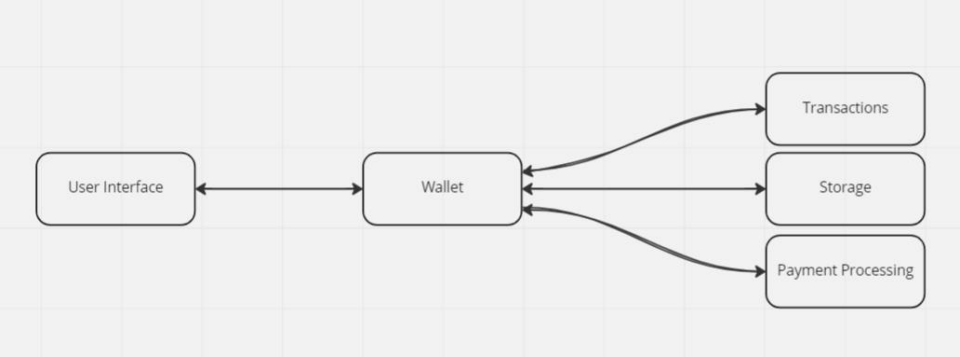

Figure 2: Conceptual Architecture for the wallet subsystem

Subsystem Concrete Architecture – Wallet

The concrete architecture of the subsystem wallet is below. While working through the architecture understanding process of propose, compare, and investigate we were able to derive the following concrete architecture. There are unforeseen dependencies that exist in the system. When looking at the concrete architecture of the wallet through the lens of the source code and visualizations created by the software Understand the wallet has a bilateral dependency on the RPC and API, Storage, Validation, and Graphical Interface. The wallet then has many other components that it does not depend on but shares information with, those

being mem pool, crypto lib, peer discovery, mining, and consensus algorithm. The wallet is comprised of two main types, nondeterministic wallet and deterministic wallet. The nondeterministic wallet is where each of the keys that exist in the user's wallet are generated. They are generated from a random number and do not relate to each other (sometime referred to as a JBOK wallet, or “Just a Bunch Of Keys”). The deterministic wallet on the other hand is where all of the keys that exist in the wallet are derived from the same seed key. This means that all of the keys that exist in the wallet have the potential to be generated if one had access to the original seed key. *A* hierarchical deterministic wallet is the most used form of deriving the proceeding keys in a deterministic wallet.

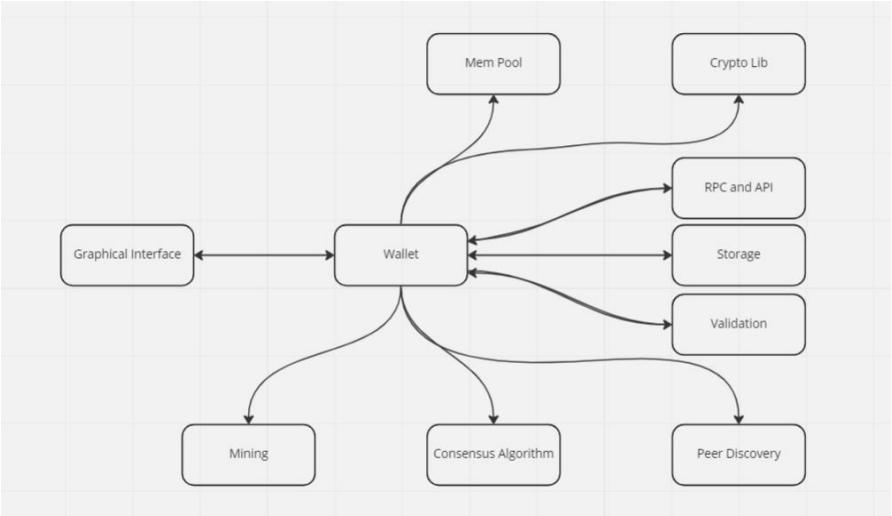

Figure 3: Concrete Architecture for the wallet subsystem

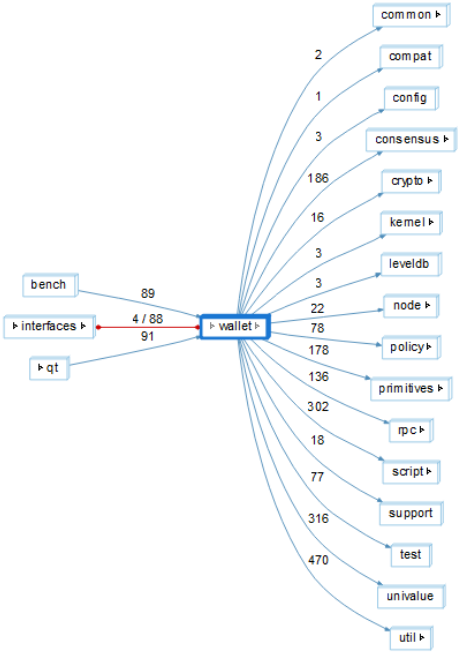

Figure 4: Wallet subsystem as shown in Understand

Reflection Analysis

After reviewing the conceptual architecture and the concrete architecture, it is clear some unexpected dependencies are present. Below is an outline of new modules that are present within the wallet subsystem, as well as dependencies that weren't shown in our conceptual architecture. These dependencies are discussed in some more detail on why they are necessary.

New Modules:

- Bench
- Qt

Unexpected Dependencies:

- Bench → Wallet
    - *Who*: Kallewoof
    - *Which*: transaction.h
    - *When*: May 1, 2019
    - *Why*: The transaction.h file contains information to add the funds from the bench component to the wallet at a given location with a set amount.
- Qt → Wallet
    - *Who*: Kallewoof
    - *Which*: isMine.h
    - *When*: Feb 13, 2019
    - *Why:* The isMine.h file contains information to check that the location of the placement of the funds is both existing, empty and owned by the owner of the wallet.

Concurrency

As discussed in the conceptual architecture report, Bitcoin Core is a multi-threaded software which uses various threads, mutexes, and global locks in order to achieve concurrency while guarding shared data structures. Much of the initialization and management of this occurs throughout /bitcoin/blob/master/src. There are many threads, such as ThreadSocketHandler, which is responsible for sending and receiving data from peers on the specified port, SchedulerThread, which is responsible for completing asynchronous background tasks such as dumping wallet contents, ThreadOpenConnections, which is responsible for initiating new concurrent connections to peers, and ThreadImport, which is responsible for loading blocks. One important thread is CCheckQueue, which handles the threads for parallel script validation for transactions in blocks. Details of this can be found in src/checkqueue.h, in which a master thread pushes verifications onto the queue of verifications that need to be performed, where they are then processed by N-1 worker threads. Here, a mutex is used to guarantee that there is only one concurrent CCheckQueueControl.

Regarding thread management for startup and shutdown, as seen in src/init.cpp, StartShutdown() sets ShutdownRequested(), which initiates the main threads WaitForShutdown(), interrupting the thread group and waiting for them to join the main thread. Once that is done, Shutdown() is called, and the system completes a clean exit. This includes flushing background callbacks in order to let the wallet catch up with the current chain more effectively and efficiently. In general, deadlocks are a possible issue of note; inconsistent locking can occur, such as if one thread locks cs_wallet and then cs_main, but a second thread locks cs_main and then cs_wallet, which is in the opposite order. This opens the system to deadlock scenarios as each thread waits for the other thread to release its lock. There are tools available to analyze and debug such lock order inconsistencies; compiling with -DDEBUG_LOCKORDER and taking a look at the debug.log file. DEBUG_LOCKCONTENTION can also be used to add additional information to the debug.log file, such as the location and length of each lock contention.

Diagrams

The first diagram represents the first use case in the Use Cases section, which depicts User A attempting to send User B bitcoin. This has been updated to reflect the divergence identified between the conceptual and concrete architecture – the wallet now makes a call to check the balance associated with the key identification number in the wallet rather than the direct balance.

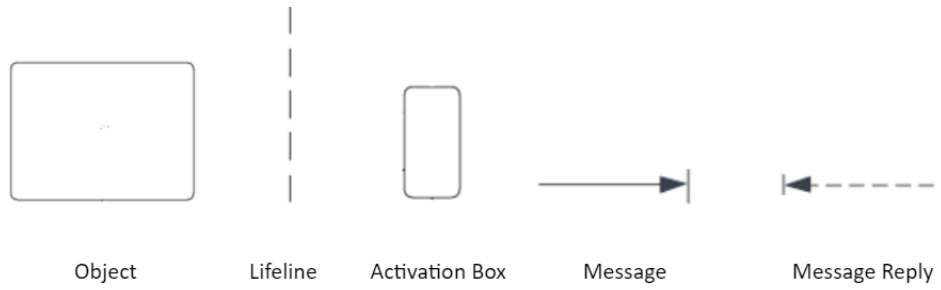

Figure 5: Sequence Diagram Legend

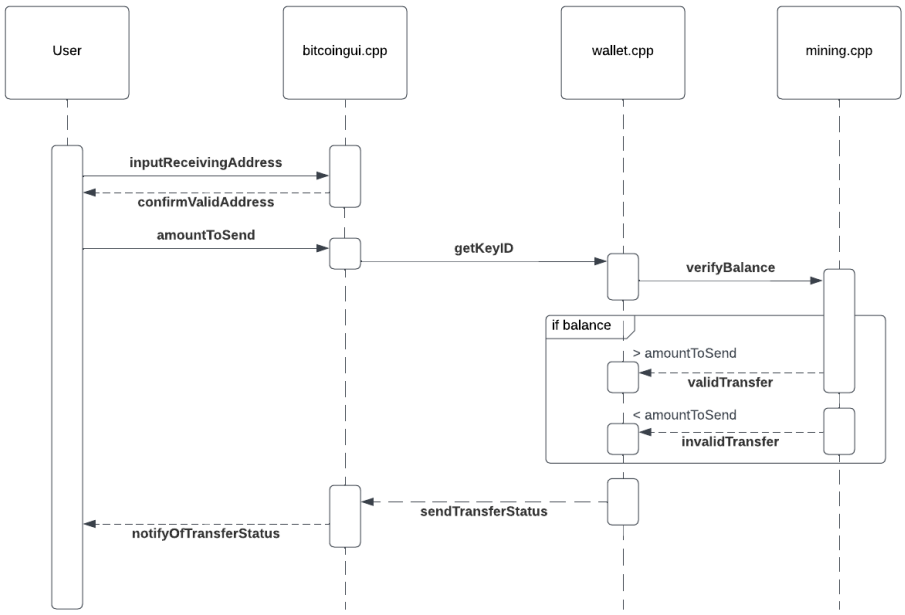

Figure 6: A sequence diagram depicts the case in which user A sends bitcoin to User B.

The second sequence diagram is identical to the one created under the conceptual architecture since no new dependencies or entities were identified.

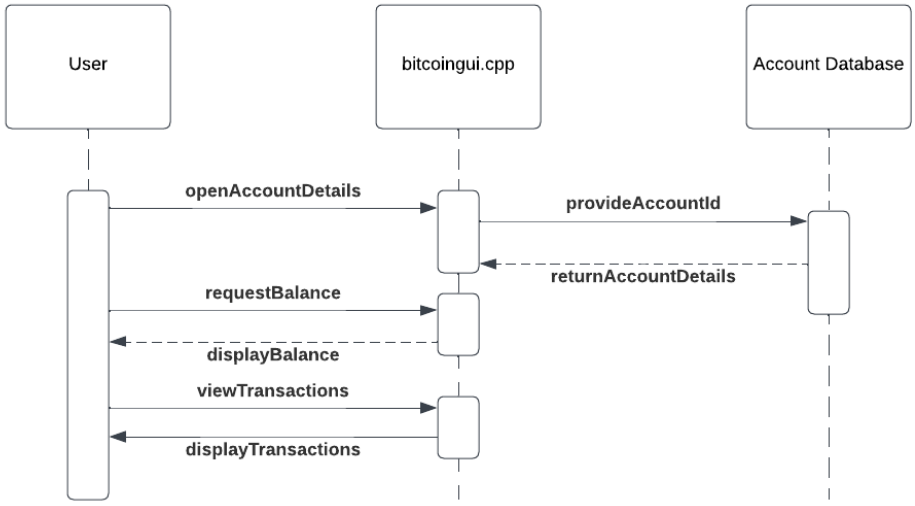

Figure 7: A sequence diagram that depicts the case where a user navigates to their settings and views their transaction history and current balance.

The third sequence diagram once again differs slightly from that created based upon the conceptual architecture. This diagram shows the presence of an isolated console function that was discovered in the source code that parses the command line input and acts. We can also

see here that the mining.cpp file is included to demonstrate the calling of this function to manipulate the node blocks.

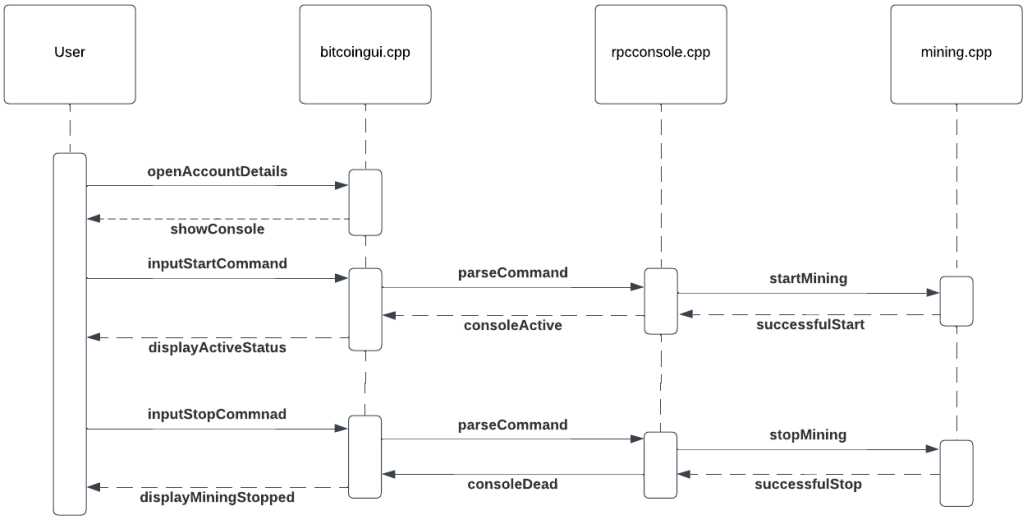

Figure 8: A sequence diagram in which the user begins and ends a mining session via the console.

External Interfaces

A number of external interfaces are present in the Bitcoin Core system. Lightweight wallets are secondary wallets for users to store their bitcoin funds in aside from the primary built-in Bitcoin Core wallet. Bitcoin Core can be used with either a graphical user interface (GUI) or command line modes, meaning information will be transmitted from the GUI to the system (mostly through functions). External connections to the bitcoin P2P network also transmit transactions, do validations, and connect to the blockchain.

Use Cases

The same three use cases that were previously identified will continue to be studied and compared to the previous conceptual architecture, which will highlight the relationships present in the concrete architecture even more.

The first use case depicts User A that would like to send a bitcoin amount to User B. This involves inputting the receiving address (User B's information), checking if the amount to send is within the spending limits of User A, and a final notice of a successful or unsuccessful transfer from the server. One large difference in the concrete architecture, which can be seen in Figure 6, is that there is no data storage in the wallet related to actual bitcoin. In the wallet, there are only keys that can be signed to verify transactions, and the bitcoin actually lives on the blockchain. By using a key to send, buy, or receive bitcoin, bitcoins themselves don't stay in the wallet of the user, but rather only the keys that verify their bitcoin.

The second use case only includes one actor, User A, that would like to check their account details, and more specifically their transaction history and current balance. The user simply interacts with the UI to show the account details. There were no divergences found in this use case since the UI is simply showing information that has already been loaded with the session, which was already accounted for.

Lastly, the third use case relates to the mining aspect of Bitcoin Core. A user can enter their account details through the UI once again but instead of focusing on transaction history, they navigate to the Console which will start or stop mining upon receiving the respective command. When evaluating the concrete architecture, it was discovered that this use case also has dependencies on the transaction module of Bitcoin Core, which makes complete sense since the work that mining does is largely verifying transactions. In addition, the console works to create node blocks and chains upon interaction, which was previously thought to be an independent actor. These changes are shown in the associated sequence diagram, Figure 8.

Data Dictionary

Mining: Users solve cryptographic hash puzzles to verify new blocks, which are then added to the blockchain ledger.

Hierarchical deterministic wallet: Generates public and private keys from one master key. The most advanced type of deterministic wallet.

Naming Conventions

API: Application Programming Interface

P2P: Peer-to-Peer architectural style

TPS: Transactions per second

SPV: Simplified Payment Verification

GUI: Graphical User Interface

RPC: Remote Procedure Call

JBOK: Just a Bunch Of Keys

Conclusion

This analysis of the Bitcoin Core architecture focused on the aspect of its concrete architecture and how it compares to and differs from its conceptual architecture. The concrete architecture was found to implement a P2P structure, which is what the conceptual architecture also proposed. Due to the new focus on the concrete architecture of the system, there were new subsystems and dependencies found that were previously unknown. Each of these dependencies were between a previously discussed subsystem and a newly discovered one.

The wallet subsystem's conceptual and concrete architectures were also discussed and compared, with the concrete architecture of a wallet being found to have 2 main types, the deterministic wallet, and the nondeterministic wallet. As previously discussed, Bitcoin Core is multi-threaded. Some of these threads were found to include the ThreadSocketHandler, CCheckQueue, and the SchedulerThread, all of which serve important roles to the software. While the concrete architecture of Bitcoin Core shows an overall well-made, evolving, and efficient system, there are still some areas for growth, including fixing the possibility of deadlock scenarios occurring when two threads lock cs_wallet and cs_main in opposite orders and are then both stuck waiting for the other to release its lock. There are tools available to debug such occurrences, but nevertheless, fixing the possibility of them happening would be an improvement to the system.

Lessons Learned

Key lessons learned include that although the conceptual architecture only included the blockchain, P2P network, wallet, transactions, mining, payment processing, and operating mode components, the concrete architecture showed the presence of a few new modules, namely the API, consensus algorithm, and cryptographic libraries. This reflection also revealed new dependencies such as wallet to RPC, validation to consensus, and transactions to crypto library. Then, additional complexities were revealed about the wallet subsystem when analyzing its concrete and conceptual architectures. Similarly to the conceptual architecture, the concrete architecture showed that Bitcoin Core uses various threads, mutexes, and global locks to achieve concurrency, also further analysis of the source code revealed critical issues relating to the potential for deadlocks within the system. Finally, the conceptualization of sequence diagrams for various use cases revealed key interactions within components of the system while exchanging bitcoins, checking transaction history and balance, and mining.

References

Bitcoin Core (2022) Bitcoin Core source code (Version 24.0.1) [source code]. https://github.com/bitcoin/bitcoin