BTC Concrete Architecture

Makayla McMullin
19mam51@queensu.ca

Aniket Mukherjee
aniketm7274@gmail.com

Daniel Dickson
d.dickson@queensu.ca

Maia Domingues
18mkd6@queensu.ca

Lucas Patoine
19lwp@queensu.ca

Devon Gough
d.gough@queensu.ca

2023-03-24

Contents

1 Abstract

The purpose of this report is to provide a technical overview of the Concrete Architecture of Bitcoin Core and compare the derived architecture with Bitcoin Core's Conceptual Architecture. Outlined in the report is the derivation process, a description of the overall Concrete Architecture of the system, as well as a detailed description of the Concrete Architecture of the Validation of Transactions and Blocks subsystem. Following that, there are reflexion analyses comparing the Concrete and Conceptual architecture of both the overall system and the chosen subsystem. Finally, the report presents two use cases that display a variety of functions explored in the Concrete Architecture and discusses lessons learned while writing the report. The overall result of the report is that the Concrete and Conceptual architectures are similar in structure, however, the Concrete architecture is more modular and contains more dependency relations than the conceptual architecture. This implies that while developing Bitcoin Core a deeper understanding of the program was acquired and some aspects were determined to be unneeded while new required dependencies were discovered.

2 Introduction & Overview

Through the examination of Bitcoin Core repositories a Concrete Architecture of the system was derived. The Concrete Architecture is composed of five subsystems: User Interface, Blockchain, Miner, Network and

Wallet. Each subsystem is connected to the others through various unidirectional or bidirectional dependencies and work together to perform Bitcoin Core functions. Comparison between the derived Concrete Architecture and the system's Conceptual architecture showed a similar architectural structure that is a mix of Peer-to-Peer architecture and Publication-and-subscription architecture. Nevertheless, there are some discrepancies between the two architectures when it comes to dependencies and inter-process communication. The Concrete Architecture of Bitcoin Core is more modular than its conceptual counterpart which uses manager modules to direct collect and direct data. This modularity was likely added by programmers to improve the program based on a deeper understanding of the system they developed while programming.

The subsystem analyzed in detail for this report is the Validation Subsystem which validates transactions and blocks before they can be used in the system. The validation system was far more omnipresent than anyone anticipated, spanning every subsystem apart from the UI. Most files use the Chainstate and ChainstateManager class defined inside 'src/validation.cpp', a few use the included mutex lock that prevents unwanted read/write access to those classes, and an even smaller portion of the files call the methods directly.

Through the utilization of use cases curtain functions of Bitcoin Core were explored using the Concrete architecture we derived. The chosen use cases of making a transaction and mining Bitcoin are very expressive of the Bitcoin Core architecture. Each case uses a wide range of subsystems to complete the required function and is a good display of the inter-subsystem communication of the system and its flow of control.

Though writing this report we gained some valuable experience analyzing large scale programs. We also deepened our understanding of cryptocurrency validation and the difference between centralized and decentralized systems.

3 Concrete Architecture

3.1 Derivation Process

In order to derive the concrete architecture, our team examined both the GitHub repositories associated with Bitcoin Core, as well as its Understand model. The Understand model helped us visualize how everything functions in relation to other components within the system, while examining the raw source code was beneficial to learn more about these specific interactions and the function of each component, both in general and in relation to other components of the system. We examined the file names first to try and figure out each repository's function, before searching through the code to figure out more specific information that would help us place each component somewhere in our visualization of Bitcoin Core's architecture. After constructing a general idea of what our concrete architecture looked like, we cross-referenced with our conceptual architecture to see the differences and similarities and further evaluate our model.

We determined that our model does fit into a Peer-to-Peer architectural model, as per our first report, while the Bitcoin Core client itself follows a style akin to the Publish-Subscribe style, instead of the Object-Oriented style we initially proposed. From this realization, we now had a reference point to continue to improve our idea of the concrete architecture of Bitcoin Core. We used models of the object oriented and peer to peer styles as a reference to modify our concrete architecture, since we determined that the system was likely modeled in these styles, and from there derived a model of our concrete architecture we are confident in. This model is essentially a blend of the two styles, to account for both the networking that goes into bitcoin transactions, as well as the construction of the Bitcoin Core client as well.

3.2 Architecture Overview

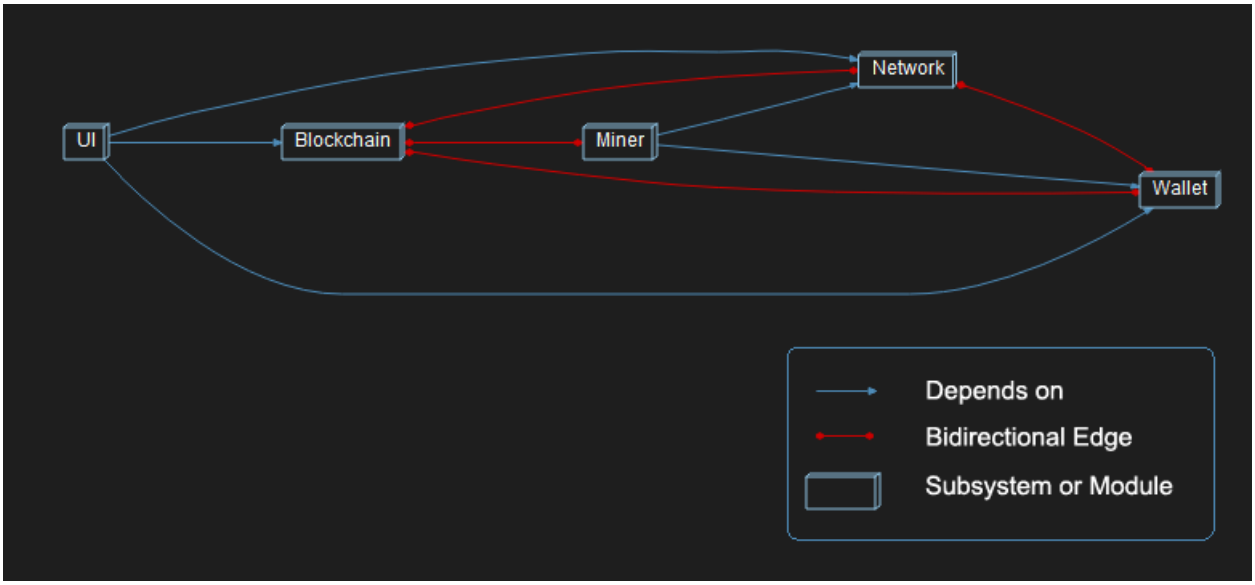


Figure 1: Understand Dependency Diagram of Bitcoin Core Architecture

We saw that the concrete architecture does appear to fit into a combination Peer-to-Peer and Publish and Subscribe architectural style fairly well. Bitcoin Core relies heavily on user action and interaction in order to function (given that it is essentially a currency system, this makes a lot of sense). It is also comprised of various components (wallet, validation, mempool, etc.) that all function in cooperation with each other, as opposed to a more streamlined approach like a Pipe and Filter style. These interactions also operate through a connector, which invokes other processes and manages the output from them. Thus, we are confident that these styles accurately portray the ways in which Bitcoin Core operates.

Each subsystem interacts with the others in some way: Miner depends on Network and Wallet, while the remainder of the relationships indicate a codependency. The Blockchain's Validation Manager (see below) serves as the system's broadcaster (a key component of a Publish-Subscribe architectural style), whereas the Network, a system that allows users to interact with each other and the blockchain itself, serves as an indicator that the system is also based on a Peer-to-Peer architectural style. In addition to these functions and dependencies, the User Interface (UI) depends on the Network, Wallet, and Blockchain to function, which makes sense as it provides the user with information about their own account and is the front-end access point that allows the user to interact with the rest of the Bitcoin Core system, including the Network and other users within it.

3.3 Blockchain Subsystem

The Mempool and Validation Engine make up the "Blockchain" part of our architecture. They each work together and have bidirectional dependencies with some of the other Blockchain components. They also have a bidirectional dependency on the Miner Subsystem, as the Miner needs to mine on the Blockchain to generate bitcoin, while the Blockchain is only functional when there are users mining for bitcoin.

3.3.1 Mempool

Mempool serves as a storage space for transactions not yet added to the blockchain. It orders these transactions from most to least desirable to mine. It works alongside the Miner Subsystem (with which it

has a codependency), as it has to tell the miners which transactions it should be mining (ie. those which it deems to be "high-priority"). It also has a codependency with Validation Subsystem, as it receives validated transactions from it to be put into the queue. The mempool is where verified blocks are stored, waiting to be mined using the proof of work algorithm so that users may gain bitcoin and add more to their wallets.

3.3.2 Validation Subsystem

The validation subsystem is a necessary component in the security of the Bitcoin system. Its job is to validate various transactions and activity that occurs within the blockchain to ensure that the system runs smoothly. It also must do this while keeping all user data anonymous, such that private user information will not be potentially exposed to the public in doing so, and user integrity is upheld. It has codependency with the Mining and Mempool Subsystems. For more information on how this subsystem functions, refer to the Subsystem Analysis section of our report below.

3.4 Bitcoin Network Subsystem

The Network is the main peer-to-peer network that allows users to interact with each other. It has a bidirectional dependency with both the Wallet and the Blockchain, as it serves as the user's primary method of interacting with the existing network of other users in order to send and receive bitcoin.

3.5 Wallet Subsystem

The wallet subsystem handles information about keys and transactions. It stores a "seed" to derive and validate private keys from using a Key Maker. The Transaction Maker keeps track of transaction requests and creates requests to other subsystem components. The Wallet Subsystem works with various other subsystems of the project. Each of its components work together to allow the user's wallet to be fully functional in managing the bitcoin that they own and the transactions they make using Bitcoin Core.

3.6 Mining Subsystem

The mining subsystem is primarily meant to assist with users generating new bitcoin. It uses cryptographic proofs to make a new block for bitcoin to be generated from. Once the block is validated in the Validation Engine, the Wallet Subsystem receives a copy of the block. The former sends the block to the Bitcoin Network to verify that the same block can not be mined by other users, and the latter sends it to the user's wallet and the local blockchain. As stated, this subsystem interacts highly with (and has a dependency on) the Wallet Subsystem, as it needs to store mined bitcoin blocks in the user's wallet, and the Validation Subsystem (with which it has a codependency), as it needs to validate transactions before allowing them to proceed.

3.7 User Interface

The user interface is responsible for obfuscating the mechanisms behind Bitcoin Core from the end user. The UI enables the user to create or import a wallet, send and receive bitcoin, and sign and verify messages. When a user goes to send bitcoin, the UI provides address validation and fee estimation. The UI is built for simplicity and ease-of-use, as it only has a few functions, starts the chain sync upon launch, and has a large help menu if a user needs assistance. It directly depends on the Blockchain, Network, and Wallet subsystems.

4 Analysed Subsystem

For Bitcoin Core to be as successful as it is, it needs a way to validate blockchain activity, as well as transactions between different sources, while every source and user is completely anonymous, for extra security measures. This is where the validation engine of bitcoin core comes into play. Every block is sent through the validation engine, and cross referenced with the network version of the blockchain, so as to make sure the block in question is valid and not spoofed in any way. The `src/consensus/validation.h` file contains an enum for both transactions and blocks specifying the various ways they can fail; blocks can fail due to invalid headers, mismatching data, missing the previous block, among other things. This file also creates a template for capturing information about block & transaction verification. Only once this validation is complete are copies of the block sent to other components of bitcoin core to carry on with other operations. The validation of transactions works in a similar fashion, as the validation engine checks if transactions are using previously unused outputs and sufficient funds to complete the transaction. Transactions can fail for having non-standard inputs, violating mempool limits, or missing witnesses to the transaction. When we examine some of the other files responsible for the validation logic, we can see that there is a lot of data being fed into the `src/validation.cpp` file, which is responsible for most of the validation logic of Bitcoin Core, used to validate blocks and transactions coming in and outgoing to other sources. Seeing as how there are a multitude of things to check blocks and transactions for to make sure they are legitimate, this does not come as a surprise in particular.

Firstly, we can see that `checkblock.cpp`, `blockencodings.cpp`, and `duplicateinputs.cpp` all directly call the `CheckBlock` method in `src/validation.cpp`. This method is used to check blocks independent of context. `src/validation.cpp` creates two classes to help with the management of the blockchain, `Chainstate` and `ChainstateManager` . `Chainstate` provides an API to update the local knowledge of the current best chain. It compares the network version to the one saved locally so that both versions match up and there are no discrepancies. `ChainstateManger` is used to interface with one or two chainstates, and enables the different states to be maintained at different block heights at the same time. Many files from different top level sub-folders use the two classes, including files from the `node, rpc, and bench` folders. Similarly, trace.h likely manages to trace back and store wallet data so that transactions between bitcoin wallets can occur correctly and securely, with a level of anonymity, which is why it feeds data into `src/validation.cpp` to begin with.

Another thing that makes a transaction valid is the transaction limit on each bitcoin transaction, there's a maximum amount of bitcoin that a transaction can be for, and if it exceeds that limit, then it's an automatic tip off to the fact that the transaction is not authentic and should be marked as invalid. `src/validation.cpp` also sends data out to other files after processing. The mutex `cs_main` is a critical lock used to guard access to the `Chainstate` and `ChainstateManager`. Major files from the `node and rpc` sub-folder check this mutex lock, including `blockchain.cpp, uxto_snapshot.cpp and mempool.cpp`, all of which are critical to bitcoin core's functionality. It also sends validation data to files handling the user interface, the usage fee rate, but most importantly, pointers to the mempool as well as files guarding access to the mempool(`mempool_entry`). As stated previously, once a block is good and validated, it can be put into the mempool along with other blocks, so that users may perform proof of work algorithms to mine these blocks and increase their balance of bitcoin by doing so.

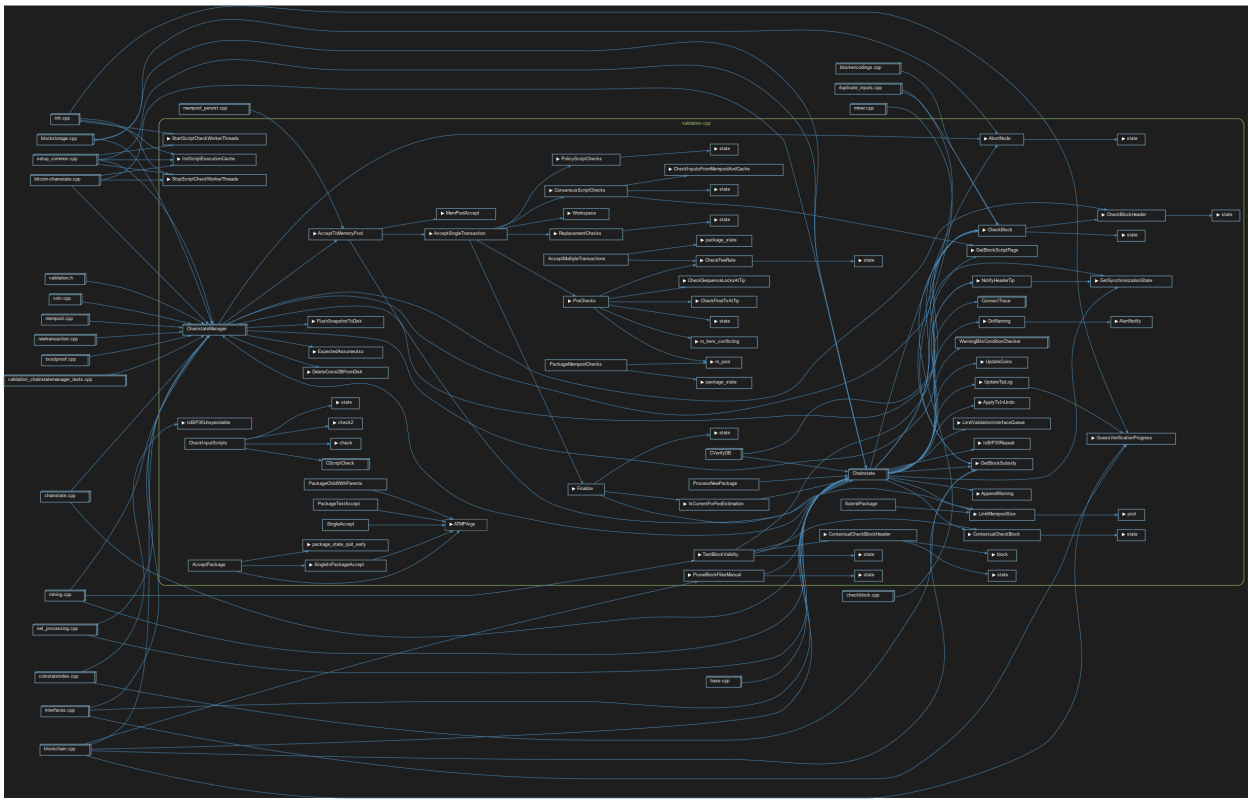


Figure 2: Understand Calls&CalledBy Diagram of Validation Subsystem

Of course, there also exists functions that stand to update an accounts coin count, and the function that does that here is dubbed UpdateCoins, so once the proof of algorithm completes, the users total number of coins will also stand to update and increase. This function doesn't only get used in this specific use case however, as it can be used in pretty much any instance where a users bitcoin balance changes, like if they buy something with it, or otherwise lose bitcoin. There are also a host of functions that `src/validation.cpp` uses to push its own data onto other files essential to bitcoin cores functionality, like the `blockvalidationstate` class mentioned earlier in `src/consensus/validation.h`.

A smaller but still relevant file is `src/script/descriptor.cpp` . This file defines a `struct` known as a descriptor, which is "a string that describes a set of `scriptPubKeys`". This `struct` contains a string that represents a set of public keys, in addition to all the information needed to decode them. This file contains a function called `PolyMod` as well as the `DescriptorChecksum` method, which checks to see that the descriptor contains valid addresses.

Additionally, there is a function to read disks for blocks, which as discussed briefly in our last report, is an option for people who want to store their bitcoin locally on a hard drive they control. In that vein there is also an accessibility function that allows the read to be cancelled halfway through in case the user decides that the read operation is taking too long for their liking. Regarding the dependencies for `src/validation.cpp`, there are 2417 of them, all of which are connected to other systems, like the chainstate system, block checking systems, and warning bits class that is designed to check conditions of blocks and data being sent through Bitcoin core, so as to maintain consistency through validity. On each block, several tests are run in accordance with all the parameters that blocks on the blockchain should be able to pass, if they are compliant with bitcoin core protocols. This adds another layer of security on top of the ones previously established, with the anonymity that Bitcoin Core affords the user.

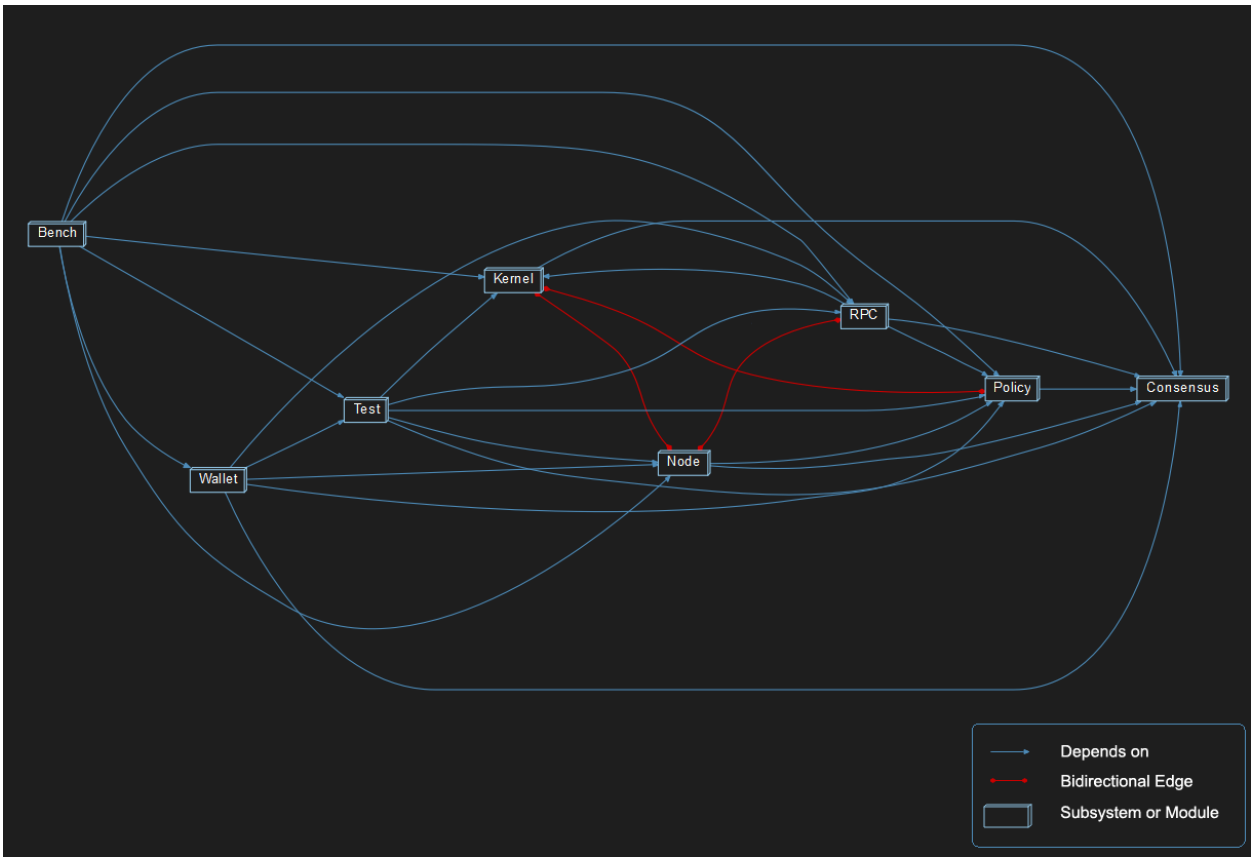
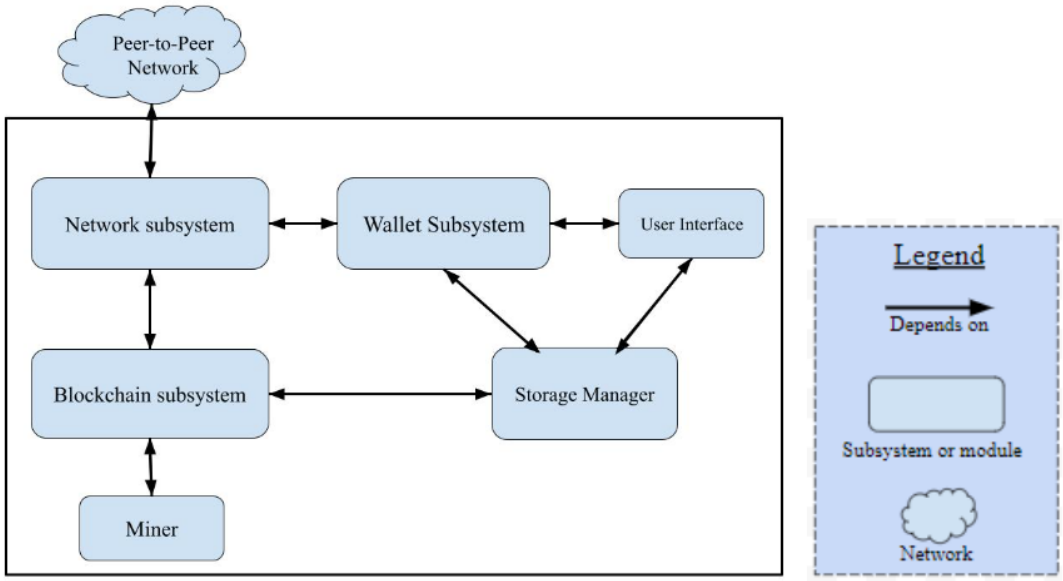


Figure 3: Understand Dependency Diagram of Validation Subsystem

5 Reflexion Analysis

5.1 Overall Architecture



The general architectural style of Peer-to-Peer mixed with Publication-and-Subscription is the same in the Concrete and Conceptual architectures. The system communicates with the Bitcoin Network via Peer-to-Peer communication, and the internal system operates using a Publication-and-Subscribe design. While the general style of architecture is the same, there are some discrepancies in the modules of the two architectures.

One of the main discrepancies we found concerned how data is stored in the system. In the Concrete Architecture, each subsystem has its own storage that interacts with the subsystem directly. Whereas in the Conceptual Architecture, there was a Storage Manager module that directs data to storage separate from the subsystems. The rationale behind this gap could be to provide more separation of concerns. Placing storage under the different subsystems instead of having it centralized makes the system more modular. Modularity helps make the system less complex and therefore easier to manage. It also provides more freedom for subsystems to have independent development and upgrades because a change in one system does require a corresponding change in another. Similarly, if there is an error or crash in one subsystem's memory the other subsystems' memory may not be affected because they are separated.

Another discrepancy between the Concrete and Conceptual architectures is the existence of an internal Connection Manager. The Conceptual Architecture contains a Connection Manager module that acts as the central hub for inter-subsystem communication. There is no such module in the Concrete Architecture; instead, the subsystems communicate directly with one another. This leads to many new dependency relations between subsystems in the Concrete Architecture such as the UI to Network and Blockchain; Miner to Network and Wallet; and Blockchain to Wallet and vice versa. Again, a possible reason for this change could be for separation of concerns and its benefits. It is also possible that programmers determined during development that the Connection Manager was just not needed for the system to operate and that time and effort could be saved by not including the module. Another reason for not including an internal Connection Manager could be to avoid making a choke point in the system. In the Conceptual Architecture, a significant amount of data flows to the Connection Manager and is directed somewhere else. This could put lots of strain on the Connection Manager and cause delays or even data loss if the system performs a high volume of tasks simultaneously. Overall, the Bitcoin Core Conceptual and Concrete architectures are very similar.

However, the Concrete Architecture is more modular than the Conceptual Architecture.

5.2 Validation of Transactions & Blocks Analysis

While performing a reflexion analysis on the validation of transaction and blocks on the blockchain within Bitcoin Core, some unexpected dependencies and gaps between the two architectures were found, as well as some similarities.

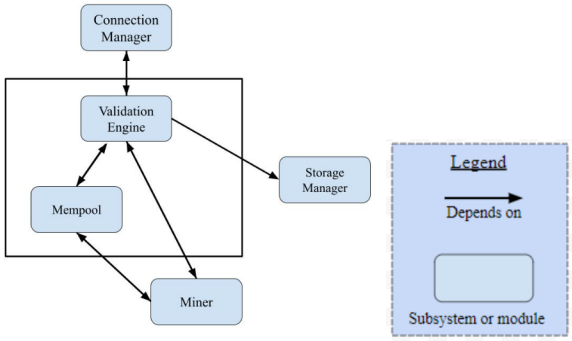


Figure 4: Conceptual Architecture of the Blockchain Subsystem, from A1

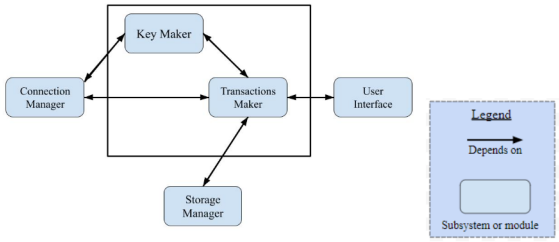


Figure 5: Conceptual Architecture of the Wallet Subsystem, from A1

- Both the conceptual architecture and the concrete architecture include a strong connection between the validation engine and the mempool.
  - Within Bitcoin Core and the blockchain subsystem, the mempool is a critical part of validation and transactions of blocks on the blockchain. The mempool has a strong dependency to the validation engine, which is apparent within both our conceptual and concrete architectures.

Discrepancies between the conceptual and concrete architecture include:

- Inclusion of a more robust validation engine architectural model, rather than including all validation components under one module as was done for the conceptual architecture.
  - Our conceptual architecture included the validation engine under the blockchain subsystem, which is in line with the concrete architecture. However, the conceptual architecture does not go into nearly as much detail as the concrete architecture does, in relation to how Bitcoin core handles validation requests within the validation engine. Within the concrete architecture, the choice was made to break up validation into its own subsystem of key components, such as the wallet interconnectivity, consensus, rpc, and node, which is described in more detail in the breakdown of the validation subsystem. This more clearly represents the interactions and dependencies between modules of the subsystem. For example, our conceptual architecture for the blockchain subsystem

does show dependencies between larger systems and the validation engine as a whole, but does not depict what other major systems connect within the validation engine itself. The choice to include validation of blocks as its own more robust subsystem within the larger, more general blockchain subsystem allows for it to be better represented as a critical piece of the architecture.

- The connection between the validation engine and wallet is made more clear in the concrete architecture compared to our conceptual architecture.
  - In the conceptual architecture, there was no clear connection included between the validation engine and the wallet, whereas in the conceptual architecture, the connection between validating transactions and accessing the wallet is made more clear. As referenced in the two diagram figures above, the wallet subsystem is separate from the blockchain subsystem with its own set of dependencies and modules. This is still true in the concrete architecture, however, access to the wallet is also found within the validation engine itself, which was not clearly defined in the conceptual architecture. Our concrete architecture dependency chart shows the wallet within the subsystem that handles validation of transactions and blocks, rather than just as a subsystem of its own, as shown in the overall Bitcoin Core architecture dependency chart.

6 Use Cases

Case 1: Making a transaction

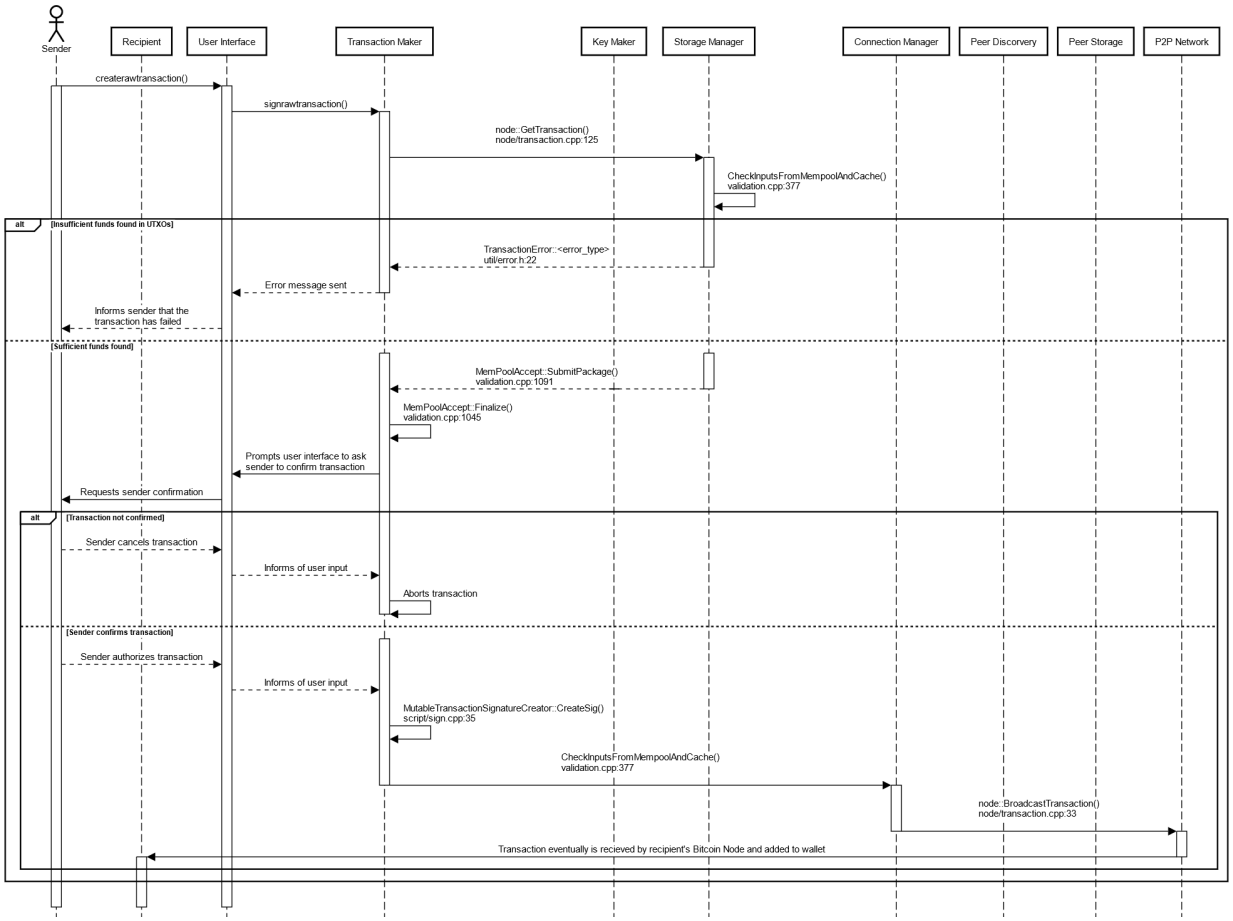


Figure 6: Use Case for User Creating & Submitting a Transaction

Case 2: Making a transaction

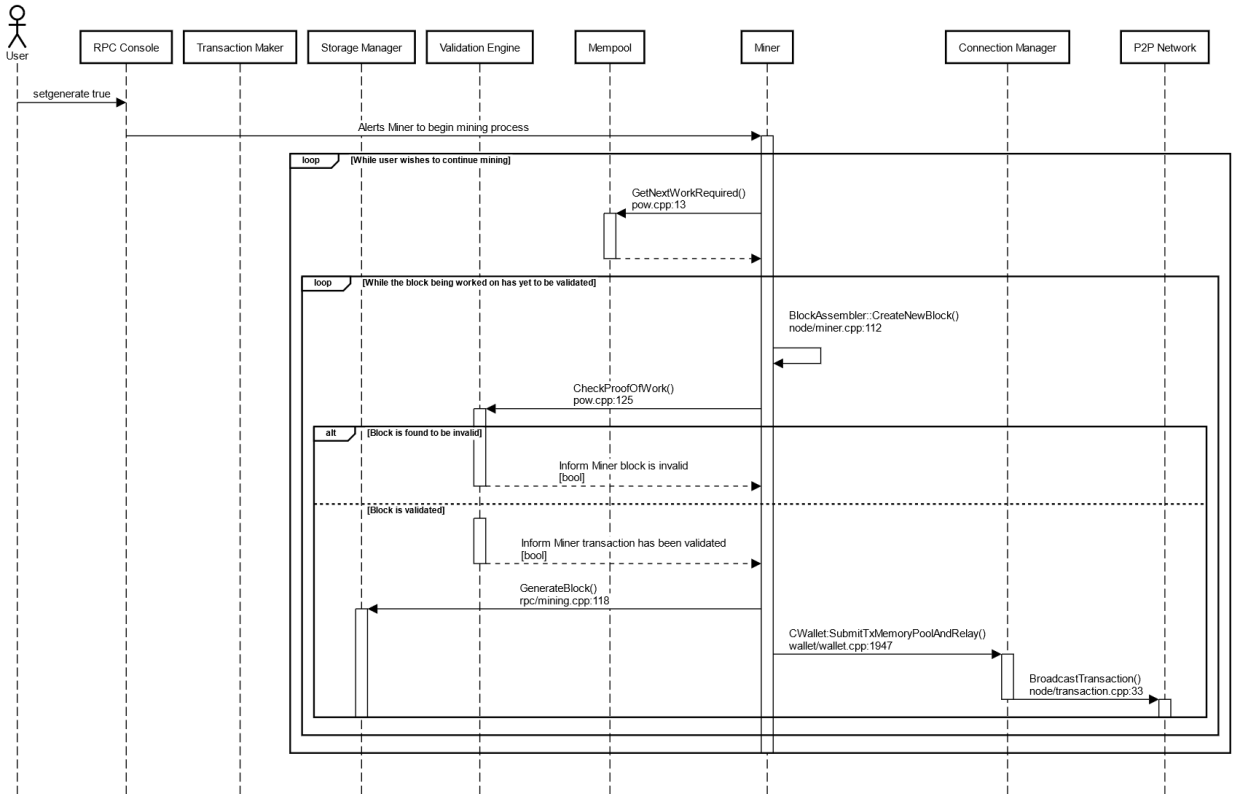


Figure 7: Use Case for Mining BTC

7 Data Dictionary

Conceptual Architecture: The view of a system's architecture based on how the system is intended to work. It is a very vague, high-level design and does not take into account implemented components of the system or its source code.

Concrete Architecture: The view of a system's architecture based on how the system is actually implemented. It is a very detailed, low-level look at how the system functions and requires analysis of the system's parts and source code to construct.

Choke point: A point in a computer system where all or the most traffic in control flow occurs.

Decentralized: A system controlled or operated by several entities rather than just one, ensuring it has no one single point of failure.

Gap: A discrepancy between a software's Conceptual and Concrete architectures.

Object-Oriented (Architectural Style): A software architectural style centered around being able to identify and protect related bodies of information or data by encapsulating them in an abstract data type. It is useful in an instance where there are many components of a project that may need to work in conjunction with each other, instead of linearly.

Peer-To-Peer (Architectural Style): A software architectural style suitable for decentralized projects

that partition tasks between standalone components that provide specific services to each other, and allow for direct interactions between each other.

Publish-and-Subscribe (Architectural Style): (also, Implicit Invocation) A software architectural style suitable for projects that involve a loosely-coupled collection of components, which carry out some operation and possibly enable other operations. A broadcasting system may invoke necessary procedures as they are needed.

Separation of concerns: A programming tactic where an application is separated into units with little to not overlapping functionality.

8 Naming Conventions

While dissecting the source code, we came across a few important files with the same name, so to differentiate between them, the path is added whenever a file is referenced in the report. In the Use Case Diagrams, a filename and line number are provided where possible for clarity.

UTXO (Unspent Transaction Output): technical term for an amount of bitcoin that in a transaction
API(Application Programming Interface): software that dictates the communication between two applications.
DNS (Domain Name System): a service that translates domain names into IP addresses.
SPV nodes: Short for simplified payment verification nodes. These note can verify payments without needing to download the entire blockchain.
Mempool: a storage space for transactions not yet added to the blockchain.
Miners: computers that add blocks to the blockchain by performing algorithmic calculations.
GUI (Graphical User Interface): what users use to interact with computer software.

9 Limitations and Lessons Learned

Our team had some difficulty accessing the documentation of the current version of Bitcoin Core when researching for this report, requiring us to rely heavily on searching through repositories, the Understand diagrams, and previous versions of the software in order to properly analyze the overall system.

We also faced some difficulty coordinating on our report due to having numerous other major assessments in other courses at around the same time as this one, and because of several group members being ill at some point during this portion of the project. We primarily worked and communicated asynchronously, but this became difficult when some of our parts depended on others' parts, or when coordinating what had and hadn't been worked on. However, this also helped people work on the project when they had the time or felt well enough to do so, as opposed to being forced to work while sick or having to neglect other priorities in order to contribute to the report and analysis. This also let us refer to each others' notes easier, which helped us to get answers to questions we may have had, keep track of whether or not we included everything in our report we intended to, and generally to help keep each other on track and on time as best as we could.

We had many positive takeaways from this project as well. Our group's newfound understanding of Bitcoin Core gave us some insight into how decentralized cryptocurrency systems function. As cryptocurrency, for better or worse, has become much more integrated into today's society, understanding how it functions can be very useful in our everyday lives and interactions with the world. Most recently, there is currently an ongoing class-action lawsuit involving many high-profile celebrities over their endorsement of a recently-collapsed centralized cryptocurrency and non-fungible token (NFT) system, called FTX. In late 2022, the system collapsed, resulting in the loss of approximately $8 billion USD worth of consumer funds [1]. Much of this was due to system insecurity and issues regarding how their system communicated with their hedge

fund, Alameda. In understanding how a secure cryptocurrency system functions, as well as the differences between centralized and decentralized systems, we can use this analysis to help us understand this scandal and apply our knowledge to the world around us to avoid putting ourselves in a similar situation in the future.

As we progress through Assignment 3, we intend to reflect on this understanding we now have, and apply it to the ways in which cryptocurrency remains relevant in the modern world to aid us in coming up with another feature to suggest. Both our system analysis and our ability to apply our knowledge to current events and situations will undoubtedly be a valuable asset to our team going forward.

10 Conclusions

In the last report, a conceptual architecture of Bitcoin Core software was proposed. One that was thought at the time to be best represented by an object-oriented architectural style, acting as both a client and server in Bitcoin's P2P architectural style. Through thorough analysis of Bitcoin Core's source code, as well as analysis of the relations and dependencies between components and subsystems through use of the tool Understand, a concrete architecture was derived. It was determined that rather than object-oriented, the Publish-Subscriber architectural style was a much better fit for Bitcoin Core itself.

Later in the report, the proposed concrete architecture was compared and contrasted to the conceptual architecture through way of reflexion analysis. It was found that while similar, there were several significant differences. One was that the proposed concrete architecture is more modular, no longer having a storage manager component and instead having it where each subsystem has its own storage. Another difference that was found is that there existed a greater number of dependencies between components in the concrete architecture. The conceptual architecture contained a connections manager component that is not present in the concrete architecture, and without it, subsystems are forced to communicate directly more frequently, explaining this difference. From this it can be concluded that as the development of Bitcoin Core progressed, it not only became more complex in nature (due to the increase in dependencies), but grew in scale as well. This would explain the removal of the storage manager, a decision that would ultimately make the system easier to manage.

11 References

1. Hern, Alex & Milmo, Dan. "What do we know so far about the collapse of crypto exchange FTX?" The Guardian. 2022, November 18 [Online]. Available: https://www.theguardian.com/technology/2022/nov/18/how-did-crypto-firm-ftx-collapse