Architectural Enhancement of Bitcoin Core

Group 20 - GoDaddy

Junhui Zong
Brian Cho
Julian Wei
Yuchen Li
Ye Shen
Arsh Kochhar

Abstract

This report discusses the implementation of a new feature which restricts node services to a subset of peers in a decentralized environment. An analysis of the current state of the architecture showed that such a feature is feasible for addition. Two ways of implementing this feature were identified. The first involves the user sharing an ID pair via a different platform (e.g. email) to the trusted peer. The second involves the creation of a new subsystem.

The SEI SAAM architectural analysis showed that both implementations involve almost the same NFRs and stakeholders, and they both affect the NFRs and stakeholders in a similar way. However, the first implementation was determined to be the better option due to it not introducing any new dependencies or subsystems at the high-level.

The nature of the feature made it so that security posed a potential risk. Developers would need to thoroughly test their implementation to identify and fix any security vulnerability. Other potential risks that were identified were performance and usability.

Introduction and Overview

The Bitcoin Core network, as a decentralized system, has played a crucial role in the growth and adoption of cryptocurrencies. While it is designed to provide secure and transparent financial transactions, there are still areas in which improvements can be made.

This report begins by introducing a suggested improvement focused on granting restricted access to node services for a select group of approved peers in the Bitcoin Core network. The subsequent section delves into the specific implementation details, offering a step-by-step guide to incorporating the enhancement into the existing network. The report then identifies the stakeholders involved and evaluates the non-functional requirements that must be met to ensure the enhancement's success.

Next, the report analyzes the realization of the proposed enhancement, discussing the impact it has on both Bitcoin Core users and developers.

Finally, two use cases are presented to demonstrate how the suggested improvement interacts with existing subsystems, showcasing the value it brings to the overall architecture of the Bitcoin Core network.

Proposed Enhancement

The proposed enhancement aims to provide limited access to node services to a subset of peers in the Bitcoin Core network. The services include:

1) **Show Keys:** Allowing authorized peers to view their own public and private keys associated with their Bitcoin addresses. This service can be useful for managing and verifying key ownership, especially for users who hold multiple Bitcoin addresses.
2) **Show Bandwidth**: Providing authorized peers with the ability to monitor their network usage, including upload and download speeds, as well as the total amount of data transferred. This service is essential for users who want to manage their network resources efficiently and optimize their node performance.
3) **Show Balance:** Granting authorized peers access to their Bitcoin balance, helping them stay informed about their holdings and enabling them to make informed decisions about transactions, investments, or trading activities.
4) **Mining Stats:** Giving authorized peers insight into their mining activities, such as hash rate, shares submitted, and estimated earnings. This service is valuable for miners who want to optimize their mining setup and monitor their mining performance.
5) **Show Spending:** Allowing authorized peers to view their transaction history, including sent and received transactions, as well as transaction fees. This service is useful for tracking spending habits, budgeting, and financial management.

Applications

The proposed enhancement has two primary applications:

1) **Bitcoin Inheritance:** The enhancement allows users to set up an inheritance plan for their Bitcoin holdings. By providing limited access to a trusted subset of peers, such as family members or legal representatives, users can ensure that their Bitcoin assets are securely and efficiently passed on to their intended beneficiaries in the event of their death or incapacitation.
2) **Backup Management:** With the proposed enhancement, users can grant limited access to their node information to a trusted subset of peers for backup purposes. This enables users to create a distributed and secure backup system for their critical node data, reducing the risk of data loss due to hardware failures, accidental deletion, or cyber attacks.

Current State of the Architecture

The current state of the Bitcoin Core architecture has undergone several modifications that enhanced its functionality. The proposed enhancement focuses on restricting access to node services for a select group of approved peers in the

network. Several subsystems in the current concrete architecture makes adding the new feature more feasible.

The API module is an essential part of the architecture as it provides an interface for other programs to interact with Bitcoin Core. Implementing the restricted access feature would involve modifying the existing API module to incorporate the necessary protocols, routine, and tools that allow for the management and authentication of approved peers.

Peer discovery contains modules responsible for discovering and connecting to other nodes in the network. The implementation of the proposed enhancement would require adjustments to these modules to ensure that only approved peers are granted access to specific node services.

The Validation Engine module is responsible for verifying blocks and transactions according to the network's consensus rules. While implementing the new feature, this module might require modifications to ensure that the restricted access rules are consistently upheld during the validation process.

Lastly, the RPC module provides remote procedure call functionality to interact with the Bitcoin Core node. Introducing the restricted access enhancement would require updates to this module to facilitate the management and authentication of the approved peers.

Implementation #1

The first implementation takes the approach of sharing secret keys between two peers via a different platform and using an authentication mechanism inspired by the BIP 150 proposal to allow services to a subset of peers [1].

The following is the sequence to limit access to the node services:

1) Integrate the public key system inspired by BIP 150 into the existing key pair generation process. This involves generating a public key and a signature for the public key to prove ownership. The corresponding private key must be kept secret and never shared.

2) Send the public key and the signature along with the public IP address via a different platform (e.g., email, fax) to the intended peer, ensuring the secure exchange of credentials.

3) Configure the visibility of services for each peer based on their corresponding peer ID, effectively controlling access to the node services.

4) Before sending a message, utilize a multi-step authentication process based on a shared secret derived from an elliptic curve key agreement protocol to establish a secure channel between the nodes.

5) When receiving messages with source (src) as the peer ID and destination (dst) as the user ID, decrypt the message, check for any tampering, and provide access to the services if the peers have the required permissions.

Impacted Subsystems

API: Create a graphical user interface (GUI) with a button to generate the key pair and the signature, and develop an API with an interface for generating these credentials. The interface to request a new ID pair must be added inside src/interfaces/node.h.

Validation Engine: Use a cryptography library to handle the encryption and decryption of messages during the generation and reception of messages.

Some functionality that will be added to the validation engine based on this process is identity verification i.e. decrypting the received message and checking the identity of the sender by validating the decrypted ID against the information stored in the Peers database, as well as message encryption i.e. encrypting the response message before sending it back to the requesting peer to ensure secure communication. A third party product should be used for encryption and decryption. Validating the decrypted ID should be implemented inside src/validation.cpp

Peer Discovery: Implement a mechanism in net_processing.cpp to detect special service messages exchanged between authorized peers and to recognize other nodes that support the authentication mechanism.

Maintain a table in the Peers database for peers with special permissions. Query the table using the decrypted ID to verify the access rights of the peer.

RPC: Provide access to the restricted services, such as show keys, show bandwidth, show balance, mining stats, and show spending, via remote procedure calls (RPC). The RPC commands should be placed inside whichever RPC file it belongs in (e.g. mining RPC commands should go in mining.cpp); however, new RPC files can be created if needed if the new command doesn't fit appropriately in any of the existing ones.

Stakeholders for the Proposed Enhancement

1) Bitcoin Core Users: These users are primarily concerned with the security of their sensitive data, as well as the benefits and value they can gain from the new feature.

2) Developers: Developers need to ensure the security of the proposed enhancement and consider its maintainability and manageability in the long run.

Non-Functional Requirements (NFRs) for Stakeholders

Bitcoin Core Users:

Security: The implementation should use encryption and authentication mechanisms to protect sensitive data during the communication process. Encrypting messages during transmission helps prevent unauthorized access and tampering, ensuring the security of sensitive data. In case a key pair is compromised, the service-providing user can simply delete it and generate a new one, further enhancing security.

Accuracy: The restricted services should return accurate values, ensuring that the data is not corrupted or changed during transmission. Using a reliable transfer protocol such as TCP can help achieve this requirement. This is particularly important when dealing with sensitive data like keys, where even a slight error can lead to significant security risks.

Performance: The response time for sending a request, receiving a request, decrypting the ID, looking up the peer in the table, determining permissions, encrypting the return message, and sending it back should be within acceptable limits (e.g., less than a specified number of seconds).

Developers:

Security: Developers must address potential legal issues and avoid causing any problems related to the security of the implementation. This includes adhering to industry best practices and relevant regulations.

Testability: Developers must run tests to ensure that this feature is secure and correctly implemented. An example test would generate the ID pair, and then use a shotgun testing method which consists of repeatedly generating a random ID and then attempting to gain access to the restricted service.

They also must ensure that more tests can be added with ease in case a new bug or a security vulnerability is found.

Lastly, the time it takes to finish these tests must be small to allow developers to be more efficient.

Evolvability: Developers must ensure that they can easily add new services.

Maintainability: The implementation should be designed in such a way that new services can be added easily, allowing for future enhancements without significant rework.

Manageability: In case of any security issues, the system should provide logs or other information to help identify and troubleshoot the problem. This will enable efficient management and resolution of issues as they arise.

Overall, the proposed enhancement positively impacts the identified non-functional requirements for both Bitcoin Core users and developers. It addresses security concerns by using encryption, authentication, and a reliable transfer protocol. Additionally, it maintains performance, maintainability, and manageability by allowing for the easy addition of new services and providing logs for troubleshooting. By carefully considering the needs of both stakeholders and adhering to these non-functional requirements, the proposed enhancement can be effectively realized in the Bitcoin Core network.

Plans for Testing

The test cases will be placed inside the src/test folder along with the other test files.

Because it is infeasible to have someone manually send messages from a separate node everytime the tests are run, the test will fake receiving a message by directly passing in messages, hardcoded in by the developers, into ProcessMessage().

To test the peer authentication implementation, a test file called “peer_authentication.cpp” will be created inside the src/test/fuzz folder. The test in this file will repeatedly generate random ID pairs and use that to attempt at gaining access to restricted services. Because the IDs are generated at random, the chance of passing the authentication test should be non-existent and the test should never grant access to the service. The test fails if access is granted for any of the requests, and the test passes if otherwise.

A second test should be created to check whether the peer is correctly authenticated by passing back in the newly generated ID pair. The test fails if access is denied for any of the requests, and the test passes if otherwise.

To test the new RPCs that gain access to the restricted services, the commands that are safe for testing will be entered inside the RPC_COMMANDS_SAFE_FOR_FUZZING list placed inside src/test/fuzz/rpc.cpp, and commands that are not safe for testing will be entered inside the RPC_COMMANDS_NOT_SAFE_FOR_FUZZING list. These tests for RPCs are more concerned with accuracy. Input partitioning, boundary value testing will be used to determine the correctness of the implementation.

Implementation #2

The second implementation is creating a new module called ACM(Access Control Manager). There are several components that will be impacted by this implementation.

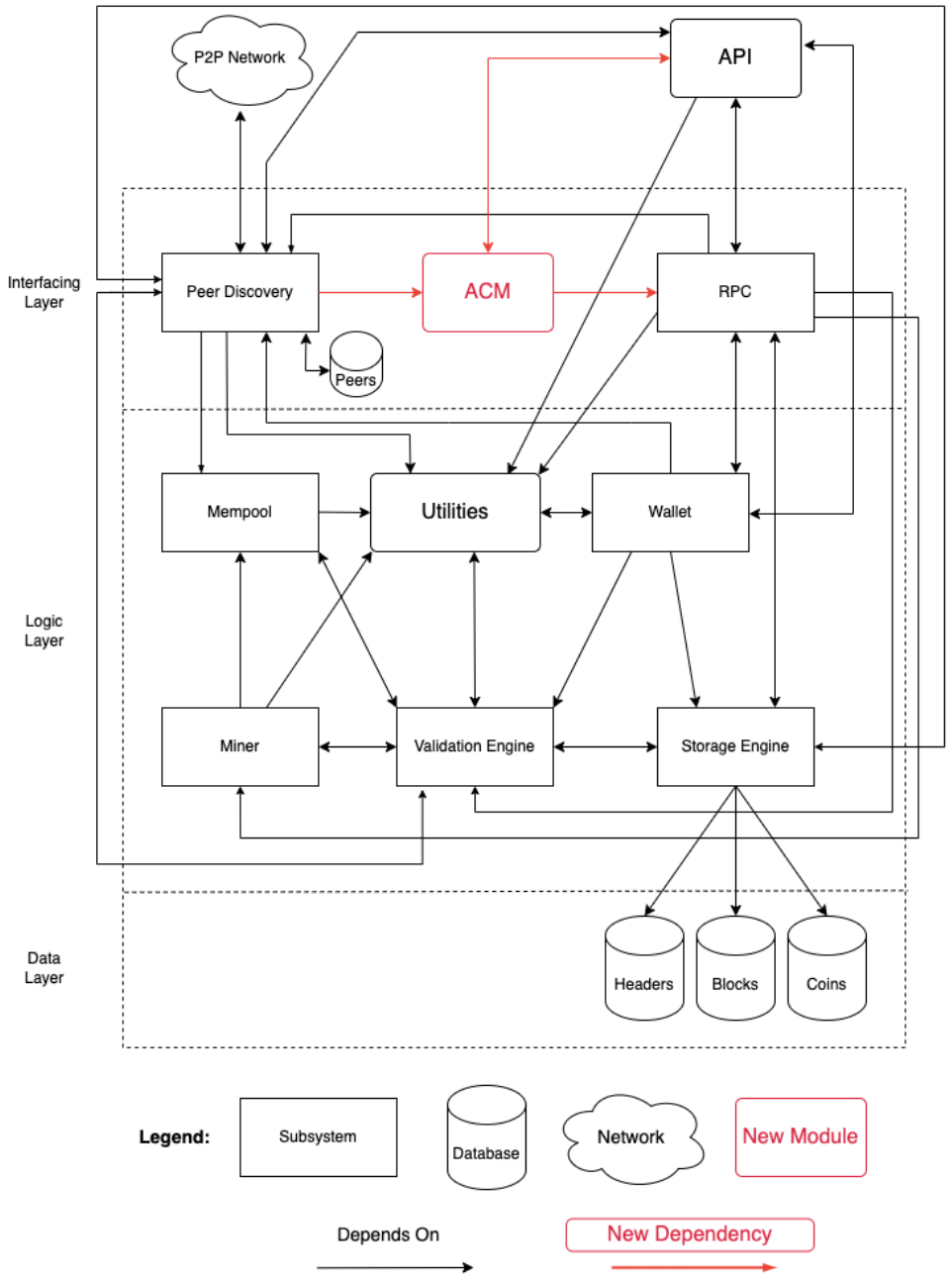

Figure 1: Concrete Architecture for Implementation 2

Impacted Subsystems

ACM (New Module): This new ACM module in Bitcoin Core will provide a flexible and extensible framework for managing Access Control across Peer Discovery, API, and RPC subsystems. Basically, the module works by providing APIs to control

access to functionalities in Peer Discovery, API, and RPC subsystems. Other than that, based on various criteria such as IP address, geographical location, reputation, or personal account information of users who are requesting APIs from ACM, we can maintain policies in ACM that can be defined to accept or reject requirements from other subsystems. When needed, these policies can be customized based on specific cases. Lastly, we may make use of external authentication and authorization systems to improve ACM as needed. And ACM will be placed in a newly created folder in src/ACM and two corresponding files - acm.cpp and acm.h will be created in that folder.

API: The Access Control Manager (ACM) module will impact the API subsystem by introducing new requirements for authentication, policy enforcement, error handling, and logging in src/API/interfaces folder. To enforce access control policies, the existing API interfaces will need to be modified to call the appropriate APIs provided by acm.cpp in the ACM module to authenticate and authorize API requests and enforce policies. The API subsystem will also need to handle errors and return appropriate error messages when requests are rejected due to policy violations. Additionally, acm.cpp in the ACM module may provide APIs for logging access control events related to API requests, which will require modification of the existing API interfaces to call these APIs to log these events appropriately.

Peer Discovery: The new Access Control Manager (ACM) module will also influence the Peer Discovery subsystem by providing a new dependency from net_processing.cpp in Peer Discovery to acm.cpp in ACM. The ACM subsystem is responsible for managing access control to the node's services and resources based on the authenticated identity of the peer, while the Peer Discovery subsystem is responsible for discovering and connecting to other peers in the network. In this case, the ACM subsystem would use the peer authentication mechanism in acm.cpp to authenticate the identity of the peer in net_processing.cpp in Peer Discovery. Once the peer's identity has been authenticated, the ACM subsystem can use its access control policies to determine which services and resources the peer is allowed to access. The Peer Discovery subsystem would be responsible for discovering and connecting to other peers that can be authenticated by the ACM subsystem.

RPC: The Access Control Manager (ACM) in Bitcoin Core will significantly impact the RPC subsystem by centralizing access control management. By introducing some functionalities such as a unified authentication and authorization mechanism in acm.cpp, ACM will control who can use RPC commands and what they can do(such as requirements from Utilities/primitives/bitcoin-wallet.cpp). Therefore, when other systems need to call the functions in the RPC placed in RPC/rpc folder, they need to go through the authentication of the ACM to restrict usage and enhance security. This will make it easier to track user activities and allow for better control over API usage.

Stakeholders for the Proposed Enhancement

1) Bitcoin Core Users: These users are primarily concerned with the security of their sensitive data, as well as the benefits and value they can gain from the new feature.
2) Developers: Developers need to ensure the security of the proposed enhancement and consider its maintainability and manageability in the long run.

Non-Functional Requirements (NFRs) for Stakeholders

Bitcoin Core Users:

Security: In the process of interacting with the ACM module, the implementation should involve strong encryption and authentication mechanisms to make sure that the privacy and security of users are protected.

Accuracy: The implementation should correctly enforce access control policies to ensure that no errors occur in the interaction between different subsystems.

Performance: The implementation should execute access control policies without affecting system performance. Users should get a response within an acceptable time.

Developers:

Security: Developers need to pay attention to possible security gaps and fix them in time to ensure the overall security of the system.

Testability: The implementation should be easy to test so that developers can identify and fix potential problems. This includes the creation of automated test suites for testing access control policies, authentication and authorisation functions.

Evolvability: The ACM module should be well expandable so that new features and access control policies can be easily added in the future as required.

Maintainability: The implementation should be easy to maintain and update so that developers can quickly resolve problems, which could improve the stability and reliability of the system

Plans for Testing

We could use unit testing here as our test approach.

The unit testing approach for the Access Control Manager (ACM) module in Bitcoin Core focuses on verifying the correctness and reliability of individual functions and components, specifically acm.cpp and acm.h. To perform unit testing, the ACM module is broken down into its components, such as authentication, authorization, policy management, and logging functions. Test cases are created to cover a wide range of possible inputs, including normal, boundary, and error conditions. Test scripts can be written using a suitable testing framework (e.g. Google Test for C++) to automate the execution of the test cases and include assertions to validate the expected output, behavior, and any exceptions. By executing tests, debugging, and fixing issues iteratively, the unit testing approach ensures a solid foundation for the ACM module, enabling it to work as intended when delivered to customers.

Evaluating the Implementations

After evaluating certain criterias, we have determined that implementation one is the overall best candidate. In this section we will discuss these criterias.

The major criteria centers on the non-functional requirements. For this both implementations are very similar in their non-functional requirements. Both have an emphasis on security and maintainability. Both when realized should be able to meet its non-functional requirements. This means that both implementations are equally valid for this criteria. Even though the major criteria passes for the both implementations, there are two other metrics which put implementation 1 on top.

The first metric will be how realistic or complete each of the implementations are. The second implementation does not clearly explain the procedures that it takes to perform some of its actions. One example of this is how the implementation handles authorization. The implementation alludes to some external authorization tool but does not specify what the tool is or how it functions. This is in contrast to implementation 1, which clearly explains its encryption and decryption process.

The second metric centers on the ease of implementation. For this implementation two has several disadvantages when compared to one. First, implementation two requires the construction of a brand new subsystem named ACM. This ACM requires multiple new dependencies to existing systems including RPC, Peer Discovery, and API. These new dependencies will need to be thoroughly tested. For implementation one, testing is still required, but the implementation builds upon the existing subsystems. No new module is required and the various subsystems it affects are all already connected to each other from previous dependencies.

After analyzing the two implementations on the criterias above, implementation came out on top due to its more complete blueprint as well as its easy to implement design.

Potential Risks

In this section we will discuss the potential risks introduced through our enhancement. We have identified three areas of concern, with it being security, performance, and usability.

For security, there are two different security concerns. First, with our implementation of the enhancement, we have given the user power to give certain peers access to its sensitive information. While this power has positives, giving users power will inevitably introduce security vulnerabilities through user errors. Imagine a scenario where the user forwards its keys incorrectly to a stranger, this stranger will then have privileged access and could access sensitive information. This vulnerability is also very difficult to eliminate as the blame lies on the user, but there could be safety nets in place to let users undo their error. The other security concern centers on a compromised privileged peer. A peer is compromised when an unauthorized third party gains access to the peer's account/node. A compromised peer will, much like the first security concern, have access to privileged actions and sensitive information of the user. This is also a difficult vulnerability to eliminate. There must be further checks to make sure privileged peers remain uncompromised.

Another area of concern is performance. Our enhancement adds new computations that the user must perform, including the encryption and decryption systems. We must make sure that these algorithms are efficient and can be scaled for a reasonable amount of peer communication traffic.

The last area of concern is usability. Our enhancement is in a way a usability improvement. We are giving the user a feature to limit certain privileged actions to authorized peers. From this, there are some foreseeable usability issues. First, there must be a procedure ready for lost keys. If the user loses its key, there must be systems in place to replace or regain the key. There must also be a system for the management of keys and peers. Users should always be able to freeze a key (render a key useless) and also remove privileged peers.

Use Case and Sequence Diagram

Use Case #1: Generating a Unique ID Pair

The user clicks on a button on the GUI that corresponds to the action of creating a new ID pair. GenerateServiceIDPair() invokes the generateidpair command in which that RPC command would tell the *Validation Engine* to generate a new ID pair. *Validation Engine* would generate both the source ID and the destination ID, and then perform checks for validity (check that the IDs are different). The ID pair would be returned and then displayed on the user's screen.

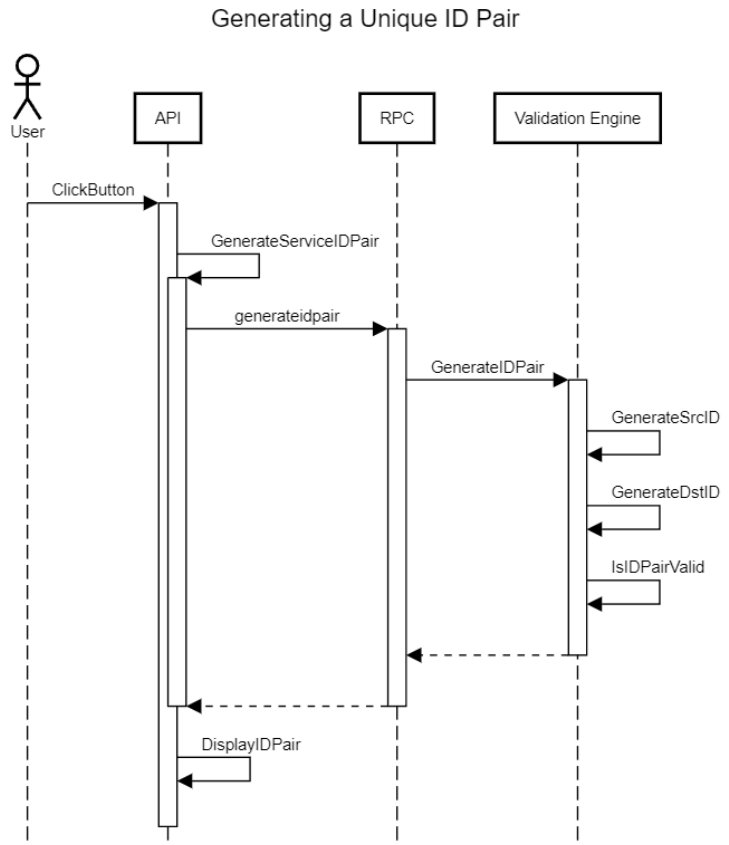

Figure 2: Sequence Diagram of generating a unique ID pair

Use Case #2: Retrieving a Peer's list of transactions

Assume that the peer has already given access permission to the user and that the peer is in the user's Peer database. Then, the user invokes the getpeertransactionlist RPC command by executing it through the RPC console. The command would ask *Peer Discovery* to send the request over to the peer. Before sending, *Peer Discovery* retrieves the ID pair, encrypts the message, and then sends it out to the peer.

The peer would listen for messages and eventually retrieve the message that the user has sent. The peer first checks the type of message and determines that it is a request for access to a special service. It decrypts the message and then authenticates the user by matching the ID pair that was embedded inside the message. Once authenticated, the peer invokes the request inside which asks for *Storage Engine* to return the transactions list. Once the list is returned, *Peer Discovery* packages the list inside a message, encrypts it, and then sends it back to the user.

The user receives the message, decrypts it, then checks that it is the message returned by the peer. The list is then displayed on the console window.

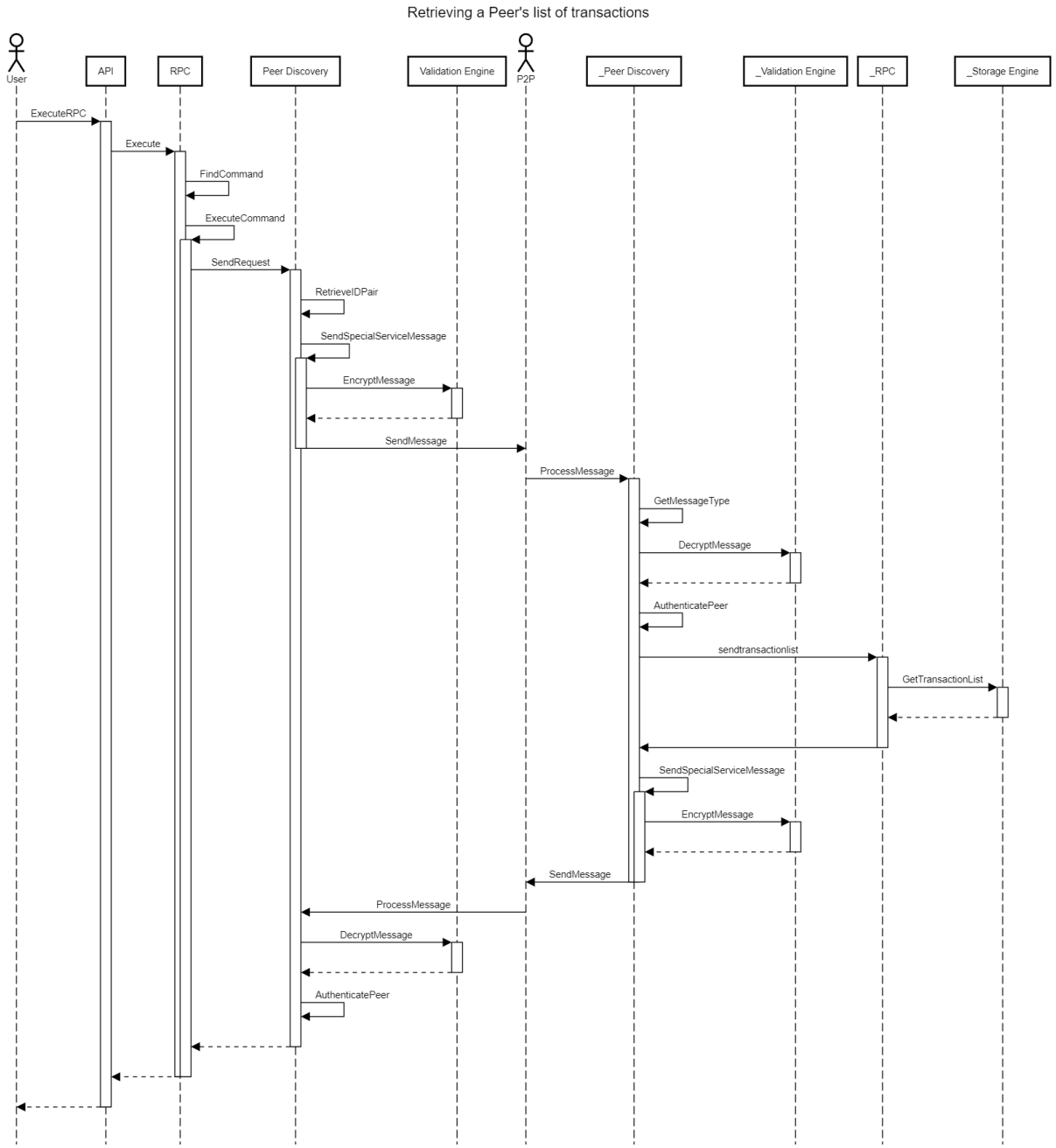

Figure 3: Sequence Diagram of retrieving a peer's list of transactions

Lessons Learned

Over the course of this assignment there were fewer issues than before. Work in the group was streamlined from experiences from past assignments. There was one major challenge though, coming up with the enhancement idea. It was a struggle as Bitcoin Core has already all its functionalities realized. There were very few actual improvements that were left. We learned that looking through the BIPs (Bitcoin improvement protocol) was one way to start thinking about improvement. Through the BIPs, we had a foundation to build upon and that is exactly what we did to BIP

150. In the future, we hope to always have a foundational piece like BIP 150 to draw upon.

Conclusion

It was found that the current Bitcoin Core architecture allows for the addition of the new feature. We were able to identify two ways of implementing the feature. The first involved the user sharing an ID pair via a different platform (e.g. email) to the trusted peer. The second involved the creation of a new subsystem.

Although both implementations involve the same NFRs and stakeholders (with the exception of the manageability NFR for implementation 1), the introduction of a new subsystem, which increases coupling, and the somewhat vague requirements of implementation 2 made implementation 1 the more ideal choice. Furthermore, implementation 1 does not introduce any new dependencies at the high-level, which makes it a more favorable choice. However, both implementations increase security risks as the new feature centers around being able to request sensitive data. Developers need to be extra cautious when implementing this feature.

Naming Conventions

RPC - Remote procedure call
BIP - Bitcoin Improvement Protocol
NFR - Non-Functional Requirements
ACM - Access Control Manager
API - Application programming interface
P2P - Peer-to-peer

References

[1] **BIP 150:** Dobson, S. (n.d.). *Bips/BIP-0150.mediawiki at master · bitcoin/bips*. GitHub. Retrieved April 10, 2023, from https://github.com/bitcoin/bips/blob/master/bip-0150.mediawiki

[2] ***Bitcoin RPC Reference:*** *RPC API reference*. Bitcoin. (n.d.). Retrieved April 10, 2023, from https://developer.bitcoin.org/reference/rpc/

[3] **Github Repo:** Bitcoin. (n.d.). *Bitcoin/Bitcoin: Bitcoin Core Integration/Staging tree*. GitHub. Retrieved April 10, 2023, from https://github.com/bitcoin/bitcoin