Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

SCIP 2.0: Horizontally Extending the Smart Contract Invocation Protocol

Akshay Patel

Course of Study:	Information Technology - M.Sc.
Examiner:	Prof. Dr. Dr. h.c. Frank Leymann
Supervisor:	Ghareeb Falazi, M.Sc.
Commenced:	September 1, 2022
Completed:	March 1, 2023

Abstract

A Blockchain is a distributed ledger composed of a network of computers that facilitates the storage and retrieval of data and keeps a record in the form of transactions. Blockchains solve double spending problems and eliminate the need for a trusted third party. Along with providing a transaction processing platform, modern programmable blockchain platforms also provide an execution environment to establish business processes through custom logic in the form of smart contracts. Recent trends show growing interest in blockchain-based applications powered by smart contracts. With a growing number of smart contract enabled blockchain platforms, executing a business process that involves interacting with multiple heterogeneous blockchains has become an intricate task. A previous work proposed the Smart Contract Invocation Protocol (SCIP), a specification that defines a homogeneous interface that encapsulates the blockchain specific interaction details to invoke a smart contract feature. An initial implementation of SCIP, referred to as SCIP Gateway, enabled interaction with Ethereum, Hyperledger Fabric, and Bitcoin. For each type of supported blockchain, the SCIP gateway has an adapter. Since the initial SCIP gateway implementation in 2019, many new smart contract enabled blockchains have been launched. This thesis proposes analysis of state of the art smart contract enabled blockchain platforms and, based on the analysis, proposes adding four new methods to SCIP and updates to the methods proposed in the previous work. Along with three new adapters for the selected blockchain platforms, as a part of the work, this thesis proposes changes in existing adapters as well. Most importantly, SCIP and SCIP gateway can now allow communication between the client application for a collaborative execution of smart contract invocation(s). Another key feature that allows developers to create new adapters in a programming language-agnostic way has been proposed in this work. Furthermore, a testing framework for executing integration tests has been developed to validate the changes for the new adapters.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

ABCI Application BlockChain Interface. 49

API Application Programming Interface. 25

BAL Blockchain Access Layer. 15

BFT Byzantine Fault Tolerance. 19

CPU Central processing unit. 20

DAO Decentralized Autonomous Organization. 44

dApp decentralized application. 35

DON Decentralized Oracles Network. 35

DPoS Delegated Proof-of-Stake. 20

HSL HyperService Programming Language. 36

NFT Non-fungible Token. 25

PBFT Practical Byzantine Fault Tolerance. 21

PoA Proof-of-Authority. 20

PoB Proof-of-Burn. 21

PoC Proof-of-Capacity. 20

PoS Proof-of-Stake. 19

PoW Proof-of-Work. 20

SCDL Smart Contract Descriptor Language. 29

SCIP Smart Contract Invocation Protocol. 15

SCL Smart Contract Locator. 29

TPS Transactions Per Second. 23

UIP Universal Inter-blockchain Protocol. 36

USM Unified State Model. 35

1 Introduction

An electronic payment system based on cryptographic proof [Mou16] was first described by an unknown person or an organization named Satoshi Nakamoto, which led to the invention of Bitcoin [Nak08]. Bitcoin was the first peer-to-peer electronic payment system that removed the need for a central authority to facilitate the exchange of value. With the advent of Bitcoin, the *Blockchain* technology has been adopted into many other application domains such as public services [ACG15], Internet-of-Things [ZW15], security services [Noy16]. Blockchain technology is becoming one of the most promising technologies for the next generation of internet interaction systems [ZXD+18]. Blockchain is a distributed ledger technology that maintains a record of transactions in an immutable manner through a consensus mechanism [Mou16]. A detailed explanation of the Blockchain technology is presented in more detail in Section 2.1.

A cryptocurrency is a digital money whose transactions are secured through cryptography [PPC15]. Bitcoin being the first such currency, provided an alternative to existing payment systems which were characterised by high transaction fees, barriers imposed through regulations, censorship, and transaction settlement spanning across days [Mou16]. Bitcoin, powered by the Blockchain technology, provides a payment system that solves the double spending problem [Nak08] and completes settlement within a short time compared to traditional payment systems, and provides censorship resistance and transparency to the users [Mou16]. Later with the need to execute custom business processes rather than just serving as a payment gateway, smart contract enabled blockchains have been developed. Ethereum[1] was the first blockchain to propose the support of *smart contracts*. The initial implementation of Ethereum used Proof-of-Work as a consensus mechanism, which required significant computing power and energy [JHGR20]. Since then, various blockchain technologies have been developed to enable faster transaction processing, and save on computation costs or even provide custom features to fit specific use case(s). Each of these blockchains has an interface that enables the users to send transactions, read data from the blockchain, or monitor events. A business process might involve interaction with multiple blockchains. When multiple heterogeneous blockchains are involved in a business process, it becomes necessary to develop separate interfacing logic for each blockchain, in order execute a complete business process. Section 2.2 describes in detail the need for blockchain integration, and Section 2.2.1 describes Smart Contract Invocation Protocol (SCIP) [FBD+20] which proposes a specification that defines an abstraction layer that can hide interaction details with each type of blockchain as a solution to the problem. The initial proposal was published in 2019 with the implementation gateway, also referred to as Blockchain Access Layer (BAL), supporting Ethereum, Bitcoin[2], and Hyperledger fabric[3].

[1] https://ethereum.org/en/
[2] https://bitcoin.org/en/
[3] https://www.hyperledger.org/use/fabric

SCIP was constructed by analyzing a set of 6 blockchain technologies that support smart contracts back in 2019. Since then, many new blockchain platforms have started supporting smart contracts or similar concepts. This thesis aims to analyze such technologies and introduce updates to the three standards/protocols to support these new smart contract platforms. The goals of this thesis work are: to select three state of the art (i) Selection of 3 state of the art smart contract enabled blockchain platforms, (ii) Analysis of the three platforms and determining the necessary changes to the existing protocols and standards, (iii) Implementing necessary changes in the BAL and new adapters for the three selected blockchains,

To achieve the goals of the thesis, the related work that provide solution on the problem of blockchain integration has been studied to understand the differences and similarities in their respective approaches. The related works that propose a solution to the problem of blockchain integration have been discussed in Chapter 3. The chapter describes three approaches: Hyperledger Cactus, Decentralized Oracle Networks, and HyperService, and their differences from SCIP and SCIP gateway.

Before selecting the any specific platform, a list of eligible platforms was created through the methodology explained in Section 4.1. During discovery process more than 20 platforms were listed where were further filtered based a list of criterion which led towards achieving the goal of selecting 3 platforms. Chapter 4 focuses on discovering platforms, defining selection criteria, and analysis criteria, and finally gives information about Aptos, Sui, Flow, and Cosmos SDK. This chapter discusses the similarities, differences, and features of the different blockchain platforms and later comments on the feasibility of integrating three such platforms to which the SCIP gateway support can be added. From the analysis of selected blockchain platforms, further chapters propose changes to SCIP specification.

The new SCIP specification is described in Chapter 5, which proposes four new *methods*, modifications to existing methods, and new error messages. A *method* is an interface that allows external consumers referred as *client applications* to interact with smart contracts [FBD+20]. Chapter 5 also describes data encoding and mapping between the format used by the SCIP protocol and the native data types for the selected platforms in Section 5.4. These proposed changes are then validated by implementing the new three adapters for selected platforms. Chapter 6 gives brief background information about the SCIP gateway implementation and changes to it. The changes to the implementation includes adding signature verification logic, support for four new methods as defined in the updated specification and also, the testing methodology. Then, chapter further proceeds to give a summary of changes in the components and URLs to the pull requests as a part of the thesis work. Chapter 6 also introduces a *generic adapter*, which aims to improve developer workflow for adding new blockchain platform support in SCIP gateway by removing the restriction to build an adapter in a particular programming language.

Finally, Chapter 7 summarizes the outcome of the work, and achieved goals of the thesis work and further possible scope of work. The outcome of the thesis work includes study of new blockchain platforms, updated specification of the SCIP protocol as per the study, three new adapters that support Aptos, Sui, and Flow platforms and the updated gateway implementation to align its working with the proposed specification.

2 Background and Motivation

This chapter lays the foundation of the research work proposed in this report. First, it describes the concept of blockchains, common terms related to it, and the composition of blockchains in Section 2.1. Later it describes the concept of blockchain integration and the incurred challenges when dealing with multiple heterogeneous blockchains. Then, as a solution to the blockchain integration problem, Section 2.2.1 discusses the SCIP protocol. Finally, Section 2.3 and Section 2.4 state the need for modifying the SCIP protocol and present the problem statement.

2.1 The Blockchain Technology

Before proceeding with the discovery of available blockchain platforms for this thesis, an understanding of the terminology, key concepts is required to perform searching using available tools such as a search engine. A blockchain is composed of different technologies like a database, a protocol to connect and communicate computers over the internet, clients that connect to it, tools for development, a component to verify the transactions (discussed in Section 2.1.1) through cryptographic proofs [RSA78], a *consensus algorithm* to reach an agreement on the state of the blockchain, and even an execution environment for executing custom code known as *smart contracts*. Based on its composition, a blockchain provides a varying degree of security, speed, and efficiency [TT19]. Therefore, to analyze a blockchain, it becomes vital to understand the components that a client interacting with the blockchain should be aware of and, thus, ultimately help to decide which features should be considered while choosing a blockchain platform. Paolo Tasca and Claudio J. Tessone presented a comparative study of blockchain platforms, deconstructed them into their building blocks in a bottom-up manner, and presented a taxonomy of the terms associated with a blockchain platform [TT19]. Based on the content presented in their research, the commonly used terms and concepts relevant to understanding a blockchain platform are discussed further.

2.1.1 Common Terminology

This section describes the terms commonly used for blockchains and their related components.

- **Digital asset**

  A digital asset is a digitized version of a product that includes specific rights to use, and typically has a value attached to it [Mou16]. E-books, images, songs, logo, and even money are some of the examples of digital assets.

- **Account**

  An account is an entity capable of holding digital assets and creating transactions. An account consist of an *address* for unique identification, a *balance* that represents native tokens that can be used to pay for transaction fees.

  For example, in Ethereum, an account can be *externally-owned* which can initiate transactions in which case it comprises public and private keys, or it can be a *contract* with code but no private key and cannot initiate transactions on its own. In the Flow[1] blockchain, an account comprises the following components: address, balance, public keys, code, and storage.

- **Transaction**

  A *transaction* is a cryptographically signed instruction originating from an account that updates the state of the blockchain. A transaction generally consists of the transaction payload, the signature of the creator, the amount of fees the creator is willing to pay, and a sequence number. A transaction payload indicates what action should be performed. For example, it can indicate the transfer of assets from one account to another, invoking some arbitrary code, or even deploying arbitrary code. Blockchains expose APIs that allow users to send transactions to the blockchain.

- **Block**

  Blocks are batches of transactions. A block consists of a blockhash derived from the transactions included in it and the blockhash of the previous block. Any change in the data in the block changes the block and invalidates the whole chain of subsequent blocks. Blocks are used to maintain a synchronized state in the whole blockchain network, which is agreed upon using a *consensus algorithm*. Typically each block references the previous block; thus, strict ordering is maintained with an exception when the forking happens. Forking is splitting of a blockchain network into multiple entities which might follow different set of rules form each other. The first block in the blockchain is called a *genesis block*.

- **Events**

  External applications might require performing some actions upon the invocation of certain logic in the blockchain. By listening to emitted *events* when processing transactions, external applications can react and perform actions, although they are not directly integrated with the blockchain.

- **Node**

  A *node* is a process that runs the blockchain protocol, executes the transactions, proposes new blocks, validates transactions from its peers connected over the network. It also stores the history of transactions and shares it with the other nodes on demand. The nodes holding the complete history of the blockchain, validate transactions are called *full nodes*. A full node can also participate in the block formation process. On the other hand, a *light node* holds only partial content of the blockchain but still is a part of the blockchain network. By not downloading the entire state of the blockchain, light nodes can operate with lower the

[1] https://flow.com/

computational resources and storage space than the full nodes. Having light nodes allows a wider range of participates to be part of the network. Section 2.1.4 gives more information about the types of nodes in blockchains.

- **Epoch**

  Blockchain platforms using Proof-of-Stake (PoS) consensus algorithm (described in Section 2.1.2) like Sui[2], Aptos[3] partition the operations into approximate or fixed non-overlapping time intervals called *Epochs*. In PoS, an epoch is used to manage the selection of validators who are responsible for validating blocks and maintaining consensus. After can epoch is over, parameters of the blockchain get updated e.g., new set of validators are selected.

- **Sequence number**

  A *sequence number* ensures that a transaction is executed only once, also referred to as *nonce*. Blockchain mitigates the risk of replay attacks by ensuring that transactions have a sequence number higher than the last sequence number used by the same account. This number also acts as a counter for the number of transactions for a given account. Using the same sequence number twice results in only one transaction getting executed and the other getting rejected.

2.1.2 Consensus algorithms

The nodes in the blockchain network have to agree on which transactions can be added to the blocks and the content of the blocks. The consensus mechanism ensures that the full transaction history is consistent amongst the peers and consistently agreed upon. The peers maintain copies of the history of the transaction to make the network crash fault-tolerant [Mou16] and available even when some peers are unavailable. As the copies of data are maintained amongst peers, information about the next block has to be communicated to peers and stored by them. The consensus algorithm defines which node can propose the next block and which transactions should be included in a block. During the consensus phase, the nodes communicate with each other to reach to an agreement on the next block. Each consensus algorithm has its advantages and disadvantages. Based on the use case, blockchains use specific consensus algorithms best fitting for the requirements. So to analyze a blockchain network, it is important to consider how the underlying algorithm maintains the state of the blockchain among the participating nodes.

In an open and distributed system like blockchains, malicious actors may try to collude and update the state of the blockchain, which harms the other participants in the network. So, a consensus algorithm has to consider the resistance to attacks and ensure the network's security. A consensus algorithm has to maintain resilience to faults in the system. In the context of distributed systems, *Byzantine Fault Tolerance (BFT)* [LSP02] is the ability of a distributed computer network to function as expected and correctly reach a sufficient agreement despite malicious actors trying to break the system or propagating incorrect information to other peers. It is important to understand how a system composed of n nodes can achieve a consensus even in the presence of m malicious nodes. Following are some of the consensus algorithms widely used by blockchain networks:

[2] https://sui.io/
[3] https://aptoslabs.com/

- **Proof-of-Work (PoW)**

  In the PoW [Nak08] algorithm, the nodes, also referred to as *peers*, compete amongst each other to solve a computation-intensive puzzle that would allow only one block to be accepted from multiple proposed blocks. The node which presents a solution first and gets approval from the peers wins and the proposed block gets added to the chain. In return, the solver gets a reward. The reward can be a blockchain-native token or some mechanism compensating for the computational resources. Bitcoin uses this consensus mechanism. Ethereum formerly used this algorithm but shifted to PoS due to the high energy use of PoW [Eth21].

- **Proof-of-Stake (PoS)**

  PoS [TT19] requires the node to stake a certain amount of collateral in the form of blockchain-native currency to participate in the consensus process. These nodes are referred to as *validators*. If a validator does not operate as per the specified protocol, then it is penalized and loses the complete staked amount or a part thereof. This process is called slashing. The validators can propose new blocks to be added to the chain and receive a certain reward for doing so. The validators can be selected randomly or through a deterministic algorithm. Compared to PoW, PoS imposes a lower barrier of entry by reducing the hardware requirements, lowering energy requirement, and reducing centralization risk. Ethereum switched from PoW to PoS because it is less energy-intensive than the former scheme [SHA+].

- **Delegated Proof-of-Stake (DPoS)**

  DPoS [BKN+21] is a variant of PoS where users holding the tokens choose a *delegate* using the voting mechanism. The delegate is then responsible for validating and executing the transactions. The voting power of the users proportional to the number token held by them. In contrast to PoW, where the node with the most computing power and PoS, where the node with the most tokens is typically selected to validate transactions and add new blocks to the blockchain, DPoS allows token holders to participate in the process of selecting delegates who will mine new blocks and reward only the most competent ones [GY22]. This enables mitigation of the risk of abuse of power in PoS system when a single token validator holds significant voting power than all other token holders. Sui is one of the blockchains using DPoS as consensus algorithm [RJCe22].

- **Proof-of-Authority (PoA)**

  A pre-defined set of nodes in this setup collaborate to accept and execute transactions. They also validate the transactions and blocks from other peer nodes. These nodes have exclusive rights to maintain the state of the blockchain. This consensus algorithm is suitable for private networks in which only selected nodes are allowed to update the state of the blockchain. These nodes act as *trusted signers* and use their private keys to sign the new blocks. Using this scheme helps to reduce the overall cost of maintaining the network by only requiring a small number of authorized nodes.

- **Proof-of-Capacity (PoC)**

  Unlike PoW, where the nodes need to allocate Central processing unit (CPU) as a compute resource for the generation of new blocks, PoC focuses on utilizing storage space for proposing the next blocks and getting rewards for it. The algorithm consists of *Plotting* where nodes participating in the consensus pre-generate chunks of data necessary for forging the

future blocks. The scheme not only has lower energy utilization than PoW, but also has a lower entry barrier in terms of costs to run nodes, as storage disks are cheaper than CPUs. SpaceMint [PKF+] and Burstcoin [4] are examples of blockchains using Proof-of-capacity as a consensus mechanism algorithm.

- **Byzantine Fault Tolerant (BFT)**

  BFT consensus algorithms use a fault-tolerant approach to reach agreement among nodes, even in the presence of malicious actors [Buc16]. If the number of failures the system can tolerate is `f`, such systems must have at least `2f + 1` processes [Buc16]. BFT is a generic term that refers to any consensus algorithm that can tolerate Byzantine faults, which are arbitrary and potentially malicious faults in the system. Practical Byzantine Fault Tolerance (PBFT) [CL99], on the other hand, is a specific implementation of BFT that was proposed in 1999. The PBFT consensus algorithm is designed to achieve consensus in a network of `n` nodes by requiring a `2/3n + 1` majority of nodes to agree on a particular transaction. Hyperledger Fabric[5] and Tendermint Core [Int15] are examples of the projects based on BFT consensus algorithm or the variants thereof.

- **Proof-of-Burn (PoB)**

  The nodes get the rights to propose blocks into the blockchain by sending some digital assets to an unspendable address in the PoB [KKZ19] consensus mechanism. The protocol generates an address to which the token can be irrevocably sent, and a verification function checks whether the address is unspendable. This mechanism is suitable for bootstrapping a new digital asset at the expense of an old asset [Gil18].

- **Hybrid**

  A hybrid scheme can be used for advanced use cases where blocks generated using Proof-of-Work act as checkpoints containing no transactions [WSW20]. The Zilliqa blockchain uses PBFT along with a round of complex computations similar to PoW for every 100^{th} block [The17].

2.1.3 Finality

Finality describes whether the proposed transaction is eventually included in a block and stored permanently. The two possible layouts for finality as described in [TT19] are:

- **Deterministic**

  In this setup, transactions are immediately confirmed/rejected in the blockchain, with consensus converging to a certainty. This property is useful for smart contracts where consistent execution is achieved across multiple nodes. Deterministic finality provides the significant advantage that application developers do not have to deal with the effects of chain

[4] https://www.burst-coin.org/
[5] https://wiki.hyperledger.org/display/fabric

reorganizations [HSLZ19]. Lamport Byzantine Fault Tolerance [LSP02], PBFT are some examples of the algorithms using which a deterministic consensus is achieved. Stellar [Ste14] blockchain is an example of a system using deterministic finality.

- **Non-Deterministic**

  In this model, the probability of disagreeing with an achieved state of the blockchain decreases over time. Such systems use randomized or inherently probabilistic consensus. As competing nodes can find multiple valid solutions simultaneously, the probability of having two or more valid states still cannot be ruled out. Such blockchain use concurrency control mechanism which attempts to rectify the outcomes of the parallel operations. For example, to minimize the probability of multiple simultaneous valid states, Bitcoin's block frequency is adjusted by changing the difficulty level of mining a block. Therefore, the overall algorithm is non-deterministic. In general, applications tend to wait for a certain number of *block confirmations*, which is waiting for a certain number of blocks to be appended to the chain, which ultimately reduces the probability of a block being overwritten by another longest chain.

2.1.4 Node types

A blockchain network can have nodes with different types of roles and responsibilities. Based on the design of the blockchain, the terminology used for referring the type of nodes changes. For example, the term *validator nodes* is typically used for blockchain systems using PoS algorithm. The blow list provides name of different types of nodes, their roles, and examples of the platforms using them.

- **Full Node**

  A full node has all the information about the state of the blockchain. A full node can process transactions, execute smart contracts, and query/serve blockchain data [Alc22]. Full nodes make the blockchains highly redundant as the information is replicated across multiple nodes.

- **Thin Node/Light client**

  Thin node, also call as *Light client* contains only partial information about the state of the blockchain rather than storing the complete state. Thin nodes only store block headers, giving them access to minimal blockchain data such as block timestamp, block hash [Alc22]. Thin nodes can connect to full nodes, fetch block information and even validate it. Running a thin node requires the least investment in hardware, operating costs [Alc22].

- **Archival node**

  Archive nodes store the same information as full nodes as well as all previous states of the blockchain [Alc22]. They are useful for querying arbitrary historical data.

2.1.5 Smart contracts

Blockchains can allow users to develop custom business logic using a programming language. These codes deployed and executed on the network are referred to as *Smart contracts*. The concept of *Smart contracts* was first introduced in 1996 [Sza96]. Smart contracts are defined as "agreements existing in the form of software code implemented on the Blockchain platform, which ensures the autonomy and self-executive nature of Smart contract terms based on a predetermined set of factors" [Sav17]. They are typically used to automate the execution of an agreement so that all participants can be immediately certain of the outcome without any intermediary's involvement. They can also automate a workflow, triggering the next action when conditions are met. A smart contract is a computer program stored in a decentralized network like a blockchain that verifies without the need for any trusted third party anywhere in the process, because the code itself is trusted. By deploying smart contracts, the conditions for the transaction to be terminated successfully or with error can be controlled. Solidity, Rust, Move, and Cadence are some programming languages available for writing smart contracts supported by platforms designed to execute the code compiled using them.

2.1.6 Scalability of blockchains

When comparing various blockchain systems, the usual metric to evaluate their performance is the number of Transactions Per Second (TPS) the system can process [TSH22]. For example, the Bitcoin network can process up to 7 TPS [CDE+16], and Ethereum accomplishes a rate of around 12 TPS [BMZ18]. Various scaling solutions have been proposed over the low TPS of blockchain systems [HHS20]. *Fist layer solutions*, which is the modification of the mechanism by which the blockchain system's network handles the distributed blockchain, such as Sharding, increasing block size, and the *Second layer solutions* by which another blockchain system runs on top of another blockchain such as State channels, Sidechains, Rollups, Validums, Plasma have been proposed [TSH22]. Blockchains can also use another blockchain as a base layer. Such setups are referred to as Layer X blockchain where X is the number of the blockchain in the hierarchy. For example, Optimism [6] is a layer two blockchain which uses Ethereum (Layer 1) as a base layer.

2.1.7 Security and Privacy

Blockchains can be configured to be open or even deployed in private environment i.e., permissioned, for deploying any custom business logic and creating/holding/transferring digital assets having real world tangible value. Such systems bring technical and operational risks which should be dealt with seriously. Any blockchain system should ensure that the state of the blockchain is modified only by valid means, even in the presence of malicious actors. To secure a blockchain, strong cryptographic mechanism should be used. Blockchains provide a certain degree of anonymity to the users because of infrastructure that allows users to transact without revealing their identity. Thus, blockchains provide privacy to users without having them give away their identities. Also, to achieve data privacy, the data can be encrypted in the transactions, which can be read only by the intended party holding the private key that decrypts the transaction data. *Data encryption* refers to

[6] https://www.optimism.io/

the cryptographic primitives used to ensure authenticity, integrity, and order of events. For example, the Bitcoin blockchain uses the ECDSA digital signature scheme for authenticity and integrity. To prove the integrity of a chunk of data, *Cryptographic Hash Functions* [AAA13], which produces a fixed-size message digest from the input data, makes it computationally infeasible to find the input that corresponds to a given message digest, and any change in input data changes the output message digest. By computing the hash of the original data and then later computing the hash of the same data again, it is possible to compare the two hash values to confirm that the data has not been tampered with or altered. The specification of Secure Hash Standard [Dan12] defines secure hash algorithms, SHA-1, SHA-224, SHA-256, SHA-384, SHA512, SHA-512/224 and SHA-512/256. SHA stands for Secure Hash Algorithm, a family of cryptographic hash functions widely used in developing applications requiring data integrity [Dan12]. Such hashing algorithms are used in blockchains to calculate hash value of the a block (i.e., *blockhash*) and the hash of each block includes the hash of the previous block, creating an unbreakable chain of blocks that is resistant to tampering or modification.

2.1.8 Other concepts related to blockchains

The previous subsections described the key terminologies and concepts related to blockchains. In addition, several other concepts are generally used while describing blockchain technologies. This subsection acknowledges these concepts as they also hold significance in comprehending blockchains. A non-exhaustive list of such concepts and terminologies is as follows:

- Consensus Network Topology types: Centralised, Decentralised and Hierarchical [TT19]
- Types of blockchains: Private, Public, and Hybrid [7]
- Transaction models [TT19]
- Token [ASZ22]
- Digital wallet [ASZ22]
- Zero-knowledge proofs [8]

2.2 Blockchain integration

Section 2.1 discussed about the composition of blockchains and related terminology. Blockchains are a realization of the concept of *Web 3.0*, which refers to the products and services offered in a decentralized manner without the need for a central authority [Woo14]. Applications built on Web 3.0 provide features like ownership of digital assets, censorship resistance, and decentralized decision-making, which are inherently not supported by *web2*. The term *web2* refers to the products and services where the ownership of assets is centrally controlled, and an entity mediates transactions based on trust.

[7] https://en.wikipedia.org/wiki/Blockchain
[8] https://ethereum.org/en/zero-knowledge-proofs/

Blockchains provide the infrastructure to deploy Web 3.0-enabled services. A blockchain can be both: a generic execution framework for development like Ethereum or an application-specific entity designed for a particular use case like Sei[9] which is designed for trading-related applications. These blockchains can exchange information and even transfer assets amongst each other through *bridges*. For example, consider Figure 2.1, where the access to exclusive artwork by an artist is controlled by owning an Non-fungible Token (NFT). NFT is defined as:"NFT is a digital asset that has a unique value in each token, as a result, it is referred to as non-fungible since it cannot be traded for other NFTs" [MMT+22]. The NFT marketplace, as shown in the diagram, is comprised of multiple blockchains. The content creator lists the artwork on the marketplace with the initial minting price on a permissionless blockchain (Blockchain 1) accessible to everyone❶. A trader monitors the network for new artwork listings, mints the NFT❷, and moves it to some other blockchain❸ where it could be sold in exchange for some tokens present only on Blockchain 2. Then, a user sees the NFT on sell, decides to buy the artwork❹, and ultimately moves it to a permissioned blockchain❺, which is used to obtain exclusive access to the artwork❻. Implementing this use case involves the interaction of multiple parties with several blockchains. Each blockchain can have its specification that allows external actors to send transactions, read the data, and subscribe to certain events. Each participant in the system should understand the blockchain interface so that bridging assets, pricing monitoring, listening to new proposed NFTs, and using the service are possible. Each type of blockchain can have its own set of Application Programming Interfaces (APIs), which adds to the effort to develop and maintain blockchain-specific requests.

[9] https://www.seinetwork.io/

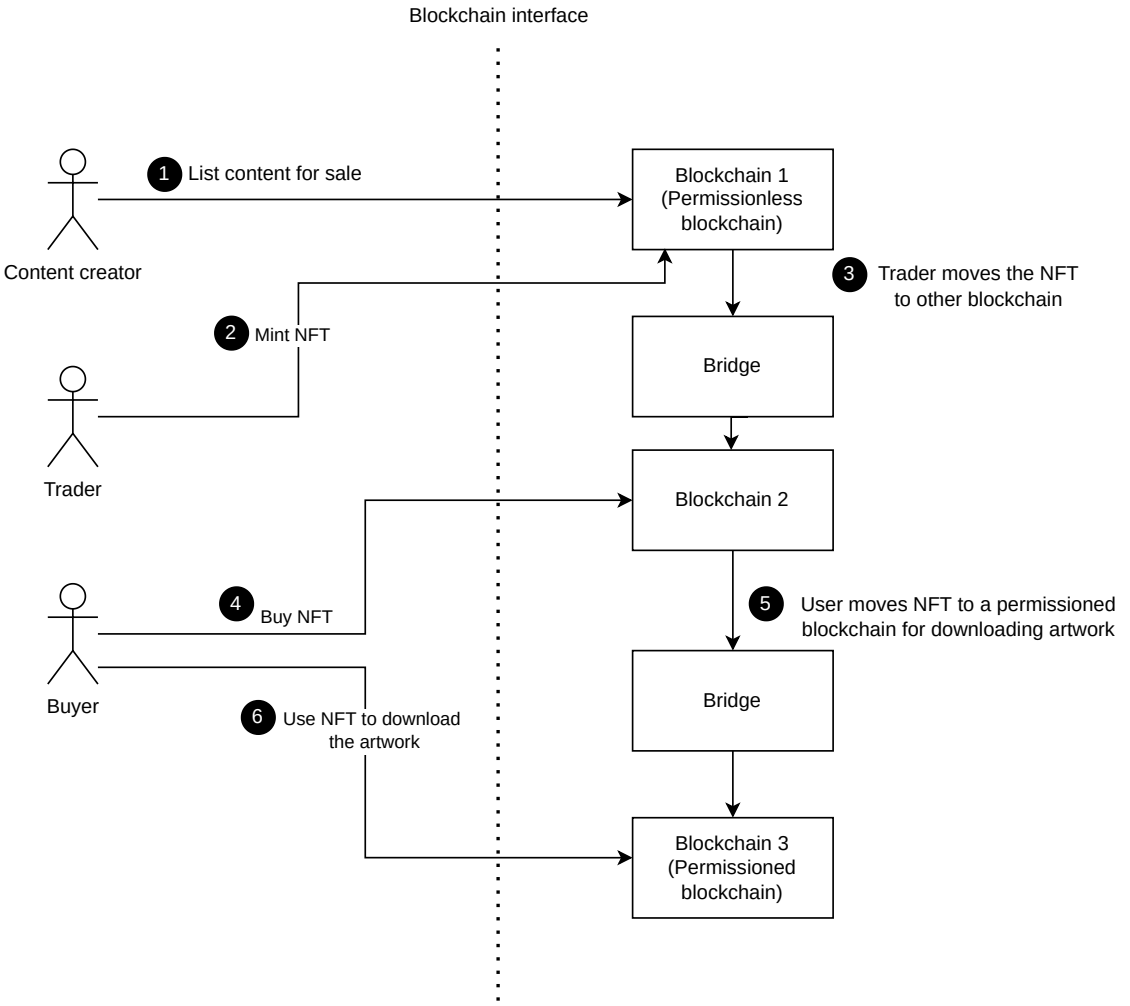

Figure 2.1: An example flow of user interaction with an application deployed across multiple blockchain

2.2.1 Introduction to SCIP

As a solution to the problem of dealing with multiple heterogeneous blockchains, SCIP [FBD+20] proposed a layer of abstraction on blockchain specific requests in the form of a protocol that would allow developers to invoke the transaction specific to a particular blockchain, query the state from a smart contract, and even subscribe and unsubscribe to the events emitted during the execution of smart contract by using a homogeneous interface. Abstracting the details of interacting with a specific blockchain simplifies the integration of business workflows dealing with multiple heterogeneous blockchains with their own APIs. SCIP defines methods, data and message formats, and error types. The entity providing concrete implementation of the methods is referred to as a *Gateway*. The first iteration of SCIP, referred to as SCIP 1.0 hereon, defined four *methods*, namely: `Invoke`, `Query`, `Subscribe`, and `Unsubscribe`. Each method consists of inputs and outputs composed of *fields* which will be discussed in Chapter 5. The format of each of the methods is shown in Figure 2.2. A brief summary of the use of each method is as follows:

- `Invoke`

  The `Invoke` method allows an external application to invoke a smart contract function. The gateway formulates a transaction that complies with the format specific to the blockchain in concern and submits it to a node on behalf of the client application. Upon successful submission of the request or the occurence of an error during submission of request, the gateway sends a synchronous response indicating a corresponding status. Upon successful submission of transaction, the gateway starts monitoring the transaction status and informs the client application about the result of execution in the form of *Callback*. The callbacks are asynchronous responses sent to the client when the transaction is confirmed with enough number of confirmations or fails to execute due to an error.

- `Query`

  The `Query` method allows a client application to search for previous invocations of functions or for events. The gateway scans the history of the blockchain and sends a synchronous response to the client application containing *Occurrences* where each occurrence corresponds to an event/function invocation.

- `Subscribe`

  The `Subscribe` method allows the client applications to receive notifications whenever a specific function is invoked or a specific event is emitted.

- `Unsubscribe`

  The `Unsubscribe` is used to cancel the subscriptions created using the `Subscribe` method.

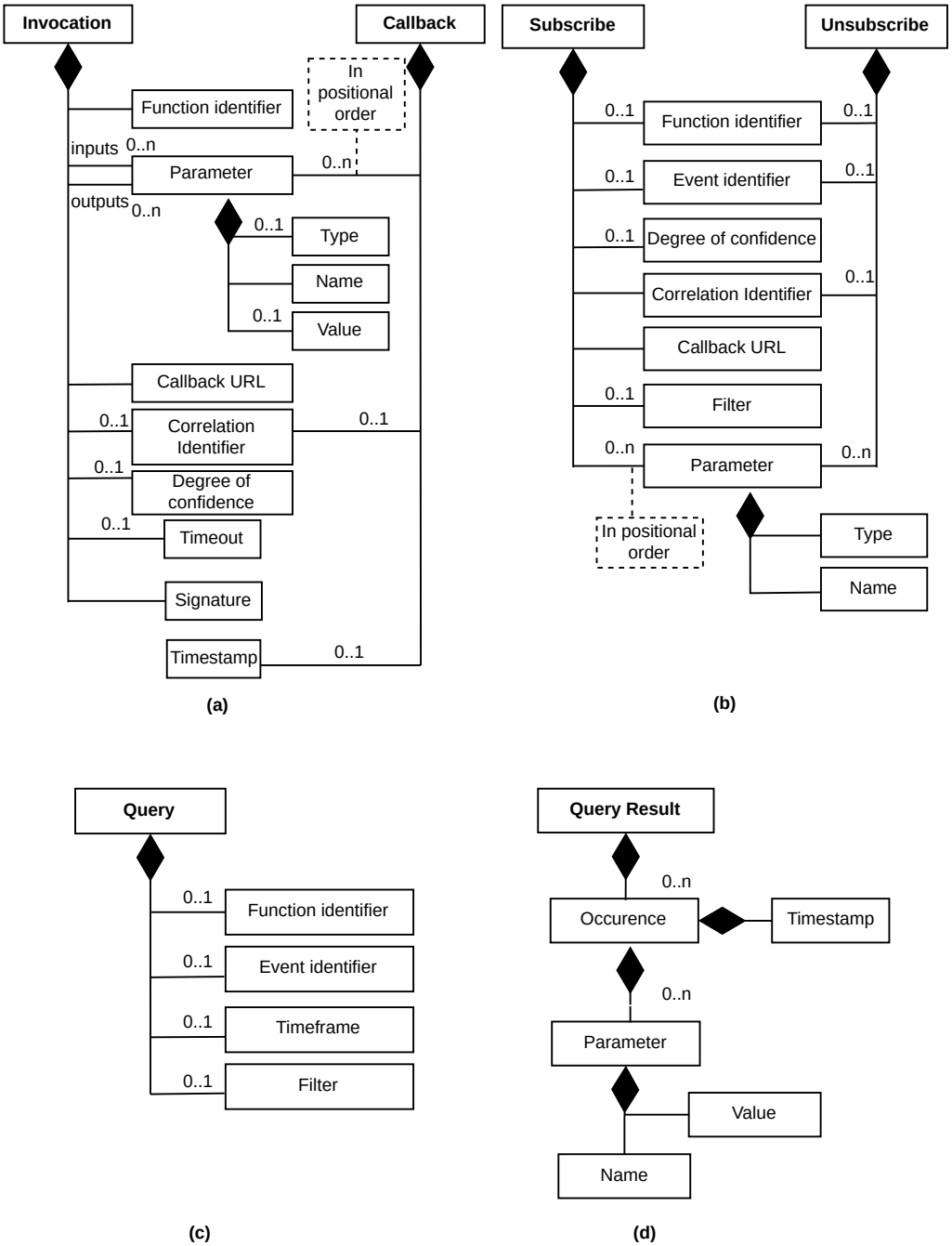

Figure 2.2: SCIP method request formats with inputs and outputs
Source: [FBD+20]

2.2.2 SCL and SCDL

Smart Contract Locator (SCL) is a smart contract addressing format for identifying a smart contract unambiguously, both externally over the internet and within the blockchain network [LFB+19]. SCL helps an external consumer to invoke a smart contract function deployed on a blockchain network to which it doesn't have direct access but may have to utilize an entity i.e., *gateway* that meditates between the blockchain node and the consumer. SCL is a specialization of a URL [BMM94] defined in Listing 2.1

Listing 2.1 SCL specification for locating a smart contract [LFB+19]

```
SCL = scheme:[userinfo@]host[:port]path"?"scl_query
scl_query = blockchain="bc"&blockchain-id="id"&address="addr"
bc = "ethereum"| "bitcoin"| "fabric"| "eosio"| ...
id = NetworkIdentifier // not further detailed here
addr = eth_addr | bit_addr | fab_addr | eos_addr | ...
eth_addr = 40ByteHexString // not further detailed here
bit_addr = Bech32Address // not further detailed here
fab_addr = PathString // not further detailed here
eos_addr = 12CharacterString // not further detailed here
```

Suppose a gateway is hosted at the domain `mygateway.com`. Some examples of accessing smart contracts for a set of supported blockchains using the https scheme using this gateway is as follows:

- Ethereum

  ```
  https://mygateway.com?blockchain=ethereum
  &blockchain-id=eth-mainnet
  &address=0x690B9A9E9aa1C9dB991C7721a92d351Db4FaC990
  ```

- Bitcoin

  ```
  https://mygateway.com?blockchain=bitcoin
  &blockchain-id=btc-mainnet
  &address=3L8Ck6bm3sve1vJGKo6Ht2k167YKSKi8TZ
  ```

Smart Contract Descriptor Language (SCDL) defines the concepts related to smart contract like variables, data types, functions and their inputs and outputs, in an abstract blockchain-independent manner to cater to external consumers [LFB+19]. SCDL is a metamodel with concrete JSON syntax. A general structure of a SCDL specification of a smart contract is show in Listing 2.2

Listing 2.2 General structure of SCDL descriptor [LFB+19]

```
{"scdl_version" : "1.0.0",    // generic smart contract properties
    "name" : "TokenConversion", ...
    "functions" : [
        { "name" : "convert", ...       // function properties
            "inputs" : [
            {   "name" : "amount",
                "type" : "number"       // list of parameters
            }, ...],
        "outputs" : [...],              // list of parameters
        "events" : [...]                // list of parameters
        }, ...                          // list of functions
    ],
    "events" : [
        { "name" : "...", ...           // event properties
            "outputs" : [...],          // list of parameters
        }, ...                          // list of events
    ]
}
```

The initial SCIP gateway prototype is reachable using SCL and the client application which is interested in invoking smart contract functions must be aware of the relevant SCDL descriptors.

2.3 Motivation

Since the first SCIP protocol specification, many new blockchains have emerged with fundamental differences in underlying working. Furthermore, the APIs exposed by these new blockchains have introduced new features that add to the security and user experience. However, the SCIP 1.0 specification has not been updated with the evolving landscape of blockchain technology. A study of new blockchain platforms by studying their composition, analysing the features of the programming language used for smart contract development, and finding the feasibility of adding support for the features of such platforms using SCIP will lead towards achieving the goal of determining what changes need to be made to the SCIP protocol.

The current SCIP specification gives power to a single client to invoke any smart contract function and limits the clients from collaborating amongst each other to invoke one single transaction. For increased security and to mitigate the risks associated with single-key wallets, multi-sig wallets have been put into practice [Bin18]. The lack of support for dealing with transactions involving approval from multiple client applications at the protocol level, hinders the use of multi-sig wallet or transactions supported by blockchains.

The prototype of SCIP gateway is currently only compatible with three blockchain technologies: Ethereum, Bitcoin, and Hyperledger fabric. This limited compatibility limits the potential for widespread adoption and usage of the prototype. A study on selection of such new platforms adding the support to the prototype would make the prototype more versatile, unlock new use cases and business workflows.

2.4 Problem statement

This research will answer following questions (RQ):

RQ1 How can blockchains be selected for analysis?

This research question aims to propose a methodology for discovery and explore the criteria for selecting blockchains for analysis.

RQ2 How to analyze blockchain technology?

This research question aims to propose the properties for studying blockchain technology and present information about the selected technologies from the perspective of these properties.

RQ3 What updates can be proposed to SCIP for it to be suitable for new blockchains?

This research question aims to propose updates to the SCIP to make it suitable for new blockchains and present the feasibility of adding support to the SCIP gateway by building prototypes of selected blockchains.

3 Related Work

This chapter provides an overview of the related work in the fields of blockchain integration. This chapter discusses three such works, namely: (i) Hyperledger Cactus (ii) Decentralized Oracle Networks (iii) HyperService , provides a brief overview of the solution for blockchain integration problem, and comments on their similarity and differences with SCIP protocol and the gateway.

3.1 Hyperledger Cactus

Hyperledger Cactus [MBH+22] is a pluggable framework for integrating multiple heterogeneous blockchains. It aims to provide an abstraction over blockchain specific protocol implementations and enable interoperability. It provides standardized interface across protocols which simplifies the interactions with multiple blockchains. The framework implementation is developed in Typescript and bundled with Webpack.

The key design principles of the framework are [Hyp22]:

- Plugin architecture

  Enhance flexibility and ensure future compatibility by utilizing a plug-in architecture. The framework allows different blockchain platforms to be integrated through the use of plugins. This makes it easier to add support for new platforms and update existing ones without having to modify the core framework.

- Secure by default

  Minimize the need for users to take explicit actions for ensuring a secure deployment of the framework.

- Toll free

  The of tokens for the transactions should be hidden from the users. At the same time, the operators should be mandated to charge fee on individual transactions.

- Low impact

  The framework should not disrupt or hinder existing network requirements.

The key components of the Hyperledger Cactus framework are:

- Business Logic Plugin

  This entity is composed of web application or smart contract in the form of single plugin, and provides integration services across multiple blockchains and executes business logic.

- CACTUS Node Server

  This entity accepts requests from external entities like end user applications willing to execute a business logic and returns an ID upon acceptance of the request.

- Validator

  This component monitors the execution of a transaction on the blockchain platform for which it has been configured. The result of execution can be failure, successful execution, or timeout, which is then signed by the validator.

- Verifier

  This component verifies and accepts the results from validators whose results are digitally signed and valid.

- Ledger Plugin

  This entity is responsible for executing a business logic on a particular blockchain for which it has been developed. It is composed of a validator and a verifier.

Hyperledger Cactus has support for the following blockchain platforms: Hyperledger Besu, Hyperledger Fabric, Geth, Corda, Quorum, Iroha, Xdai, and Hyperledger Sawtooth.

Hyperledger Cactus is a complete framework that defines the components, roles, and message flow for a blockchain integration platform. At the same time, SCIP protocol is an abstract specification for inputs and outputs for invoking a smart contract and with detailed specifications of in-between the components involved in the process. SCIP provides a more versatile API to handle function overloading. Moreover, the newly proposed specification can handle multiple signature transactions, specifying type arguments in the invocation, canceling, and replacing the invocations. SCIP gateway implementation is similar to a certain extent to the architecture of Hyperledger Cactus, as both of them support extending the implementation through plugins. However, SCIP provides a plugin-based capability add-on only for adding support for new blockchain platforms, whereas Hyperledger Cactus supports adding plugins for other functions as well.

3.2 Decentralized Oracle Networks

An *oracle* is defined as follows: "Blockchain oracles are entities that connect blockchains to external systems, thereby enabling smart contracts to execute based upon inputs and outputs from the real world."[Cha21]. Blockchain platforms cannot directly fetch data from external sources like a REST API service because of the consensus mechanisms underpinning blockchains. Many use cases, like the outcome of a sports match, weather data, prices of products, and smart contracts, require data from external sources. In such cases, oracles act as a data source and feed it into smart contracts. However, relying on proving the authenticity of data provided by the oracle is a challenge. To solve this problem, Chainlink conceptualized a fully decentralized network of oracles, also referred to as *oracle networks* [SN17]. By taking a decentralized approach, oracle networks limit the trust in a single party for providing required data from external sources. Oracle nodes, run by multiple entities, relay data to the smart contracts in the form of *reports*. A group of such oracle nodes is called a *committee*.

The second iteration, called Chainlink 2.0, led to the foundation of Decentralized Oracles Networks (DONs) [BCC+21]. DONs aims to mediate between a blockchain platform and an off-chain system by offering extensible, flexible adapters. A committee of Oracle nodes maintains a DON which decides what oracle function to execute. Thus, a DON acts as a blockchain abstraction layer that interfaces off-chain services to smart contracts and other systems. A DON consists of *executables* and *adapters*. An executable is a program that runs continuously, and an adapter acts as a bidirectional connector with the blockchain. For example, the Figure 3.1 shows a basic DON fetching data from external sources and relaying it to a smart contract using another adapter.

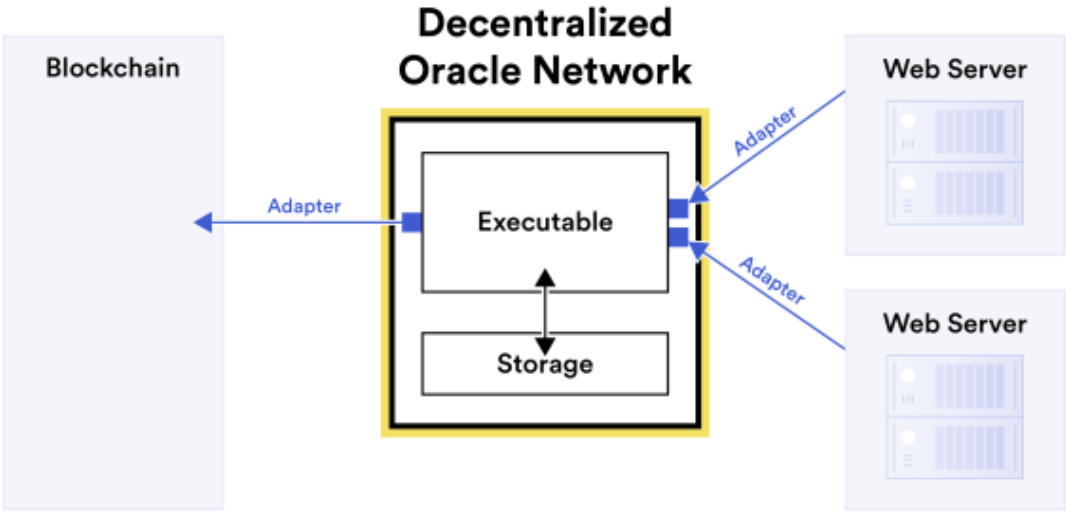

Figure 3.1: Chainlink: DON using adapters to fetch data from external sources and feeding into blockchain
Source: https://research.chain.link/whitepaper-v2.pdf

One of the key goals of Chainlink 2.0 is abstracting away the complexity of the components behind DONs and making it developer friendly. Chainlink 2.0 further aims at providing a *decentralized metalayer* wherein developers seamlessly interact with multiple blockchains which having to deal with specific details like fees, chain re-organizations, invocation interfaces.

DONs are a decentralized networks that interact with other distributed systems whereas, SCIP is an abstract specification that can operated as a single centralized permissioned entity or can be also a decentralized group of entities collaborating with each other.

3.3 HyperService

HyperService [LXS+19] is a platform for developing and executing decentralized applications (dApps) across multiple blockchain networks. A dApp is a software application that runs on a decentralized network, typically using blockchain technology. It features a programming framework and a cryptography protocol to secure cross-chain dApps. The framework includes a *Unified State Model (USM)* for describing cross-chain dApps in a blockchain-independent manner and a

high-level programming language called *HyperService Programming Language (HSL)*. The HSL code is compiled into HyperService executables and executed through the cryptography protocol. HyperService offers both interoperability and programmability across heterogeneous blockchains and includes a virtualization layer to simplify the development of cross-chain dApps. The platform also includes a *Universal Inter-blockchain Protocol (UIP)*, a cryptography protocol, to handle complex cross-chain operations securely. A prototype to prove the feasibility of the concepts incorporates Ethereum and a permissioned blockchain built using Tendermint [Int15] consensus engine.

SCIP and HyperService, although both aim to achieve interoperability, differ in their approaches for realizing the solution. SCIP does not provide a specification or a framework to develop applications but rather focuses on abstracting the invocation logic across multiple heterogeneous blockchain platforms. With SCIP already existing, deployed smart contracts can be invoked, or even their state can be read. Whereas, HyperService introduces its own development framework and mandates the use of its own programming language.

4 An analysis of existing blockchain platforms and SDK(s)

The previous chapters introduce the proposed work, outlining its motivation, the problem statement, and related works that propose solutions relevant to the problem statement. This chapter delves deeper into the analysis of different blockchain technologies by discussing the criteria used to evaluate and compare them. By using these criteria, an objective analysis of various blockchain platforms, as outlined in Section 4.1. Such an analysis would help list the features that could be added to the SCIP protocol and if these platforms support multi-signature transaction invocations. Also, understanding these platforms is a necessary step for creating adapters for the SCIP gateway.

After laying out the criteria for analysis, the chapter will discuss four specific blockchain platforms: Aptos, Sui, Flow, and Cosmos SDK. Each of these platforms will be analyzed and evaluated based on the criteria established in the previous section.

Finally, based on the analysis of these platforms, the chapter will propose updates to the first SCIP specification and provide information on the updates made to the prototype, in Chapter 5 and Chapter 6 respectively.

4.1 Blockchain Search and Selection Method

For each criterion discussed further, their importance and relevance have been provided. Furthermore, examples of some blockchain platforms that qualify or do not qualify for criteria requirements have been mentioned. To start with discovering the list of blockchain platforms that could be considered for analysis, various websites and search engines like Google[1], Bing[2] were used. The search keywords used were: *blockchain*, *list of blockchain platforms*, and then using snowballing, the links that further lead to other sources of information were also considered. During the search, after discovering a platform, its competitors were discovered, for example using keyword *ethereum competitor*. The same technique was used to search for relevant content on websites with search support like Wikipedia[3]. One of the references from Wikipedia used *List of blockchains*[Wik23] which provided names of 44 blockchain platforms at the time of writing the report. Out of these platforms, as a first filter, only those platforms (22 out of 44) have been considered that provided support for executing custom programs in the form of smart contracts. Websites like Coindesk[4] provide a dedicated section on the events, news, and innovations related to blockchain technology.

[1] https://www.google.com/
[2] https://www.bing.com/
[3] https://wikipedia.org/
[4] https://www.coindesk.com/

This discovery process was performed during the initial phase of the thesis work and terminated after two weeks as a time-bound activity. A summary of these platforms and their suitability for analysis based on the criterion is shown in Table 4.1. A threat to the validity of the mentioned method includes that the exact results might not be reproduced again because of changing search algorithms used by the search engines and the use of the Internet as a general search tool that evolves with time.

After defining the search method and listing potential platforms for analysis, the content further discusses the key points for shortlisting the platforms. The shortlisting is required so that the platforms can be studied within the time frame of the thesis work, and the goal of selecting three state of the art platforms is accomplished. The reason behind selecting these specific key points is explained further. The overall intuition behind choosing these specific properties is to improve the usefulness of the SCIP and the SCIP gateway.

- **Variety of platforms**

  The goal of the research work in this thesis is to enable the implementation of new business processes across multiple heterogeneous blockchains by supporting a wide range of platforms uniquely designed and built for specific use cases. To achieve this, the SCIP 2.0 gateway must integrate with a diverse set of blockchain platforms, each with its distinct design and implementation.

  Although many blockchain platforms and scaling solutions bring innovative approaches for low-cost transactions and faster finality without compromising security, they have not been considered for integration into the SCIP gateway or for analysis in this work. This is because many of these platforms are derivatives of an existing blockchain platform, for which the SCIP gateway already has a plugin. However, this does not mean that these platforms are not valuable; they could be considered in future work. For example, platforms like Arbitrum[5], and Optimism[6] use Ethereum Virtual Machine as the underlying technology for which SCIP gateway already has integration. Such platforms are referred to as *sidechains* or, in general, Layer 2 scaling solutions [TSH22]. Applying the same criteria, platforms like Polygon[7], Neon [8] are not considered here because they are also based on EVM.

- **Developer friendliness**

  A study on the value of software documentation has identified six quality attributes considered important for software documentation: Accuracy, Clarity, Readability, Structuredness, and Understandability [PDS14]. These attributes are critical for ensuring that software documentation is effective in helping developers understand and use the software. In the context of this report, all of the platforms discussed have documentation that meets the quality mentioned above measurement attributes at the time of writing this report.

  Platforms like Sui, Aptos, Flow and Cosmos SDK provide all the necessary guides for the developers to understand the concepts, clear and adequate documentation of APIs to interact with the platforms, and required practical tutorials to develop an understanding

[5] https://arbitrum.io/
[6] https://www.optimism.io/
[7] https://polygon.technology/
[8] https://neon-labs.org/

of the technology at the time of writing this report. On the other hand, the platform Cosmwasm [Cos21] has not been discussed in this report because its documentation was still in the alpha stage and did not include the required APIs for developing a plugin for the SCIP gateway. Therefore, the documentation was insufficient to fully evaluate the platform's capabilities or suitability for integration into the SCIP gateway. However, this is not to say that Cosmwasm can never be integrated with the SCIP gateway; it could be considered in future work.

- **Availability of APIs**

  For a blockchain platform to be supported by the SCIP gateway, it must expose an Application Programming Interface (API) that can be called to invoke smart contracts on that platform, query events, and scan for new blocks and the transactions included in them. Aptos, Sui, and Flow provide such APIs, making it possible for the SCIP gateway to interact with these platforms. However, Cosmos SDK operates differently, as it is a framework that expects the client to encode transaction data rather than providing a pre-built API.

  This means that to create a transaction using Cosmos SDK, the client must have access to the custom encoders that will serialize the transaction into a format accepted by the exposed API[9]. This can be a complex process as the client must be aware of all the primary and derived data types to encode the transaction properly. Compared to the other blockchain platforms meeting the compatibility criteria in this report, the complexity of encoding transactions from the SCIP inputs in Cosmos SDK goes beyond the scope of this thesis's work. However, the report includes the study of Cosmos SDK because a plugin could be built for a specific blockchain created using Cosmos SDK rather than Cosmos SDK itself. The blockchain created using Cosmos SDK would have specific use cases and provide the necessary encoders to create the transactions. These specific encoders can be used to create a plugin for the SCIP gateway.

- **Use cases**

  Another important aspect to consider while selecting the blockchain for integration with SCIP is that it is not limited to cryptocurrency applications but can be used for various purposes. Ideally, although blockchain could be designed for a specific purpose, it should offer the possibility of deploying general-purpose applications. Therefore, the platform's architecture and interaction tools, like SDKs, should allow developers to develop and deploy any use case rather than being restricted to one particular use case of cryptocurrency and payments.

  For example, Litecoin[10] and Peercoin[11] do not satisfy this criterion.

- **Programming language**

  Programming languages in Blockchain enable developers to develop smart contracts that are used to execute arbitrary business logic. A smart contract can be developed using general-purpose programming languages like Java, C++, Python, Golang, and Blockchain-specific languages like Vyper and Solidity for Ethereum, Move for Sui and Aptos, Cadence for Flow, WebAssembly for Cosmwasm. The code written using these programming languages is

[9] https://docs.cosmos.network/v0.47/core/encoding
[10] https://www.litecoin.net/
[11] https://www.peercoin.net/

usually compiled into low-level instructions that are then executed by a *virtual machine*, i.e., a component of blockchain nodes. Most of these programming languages offer general features such as functions that either modify the state of the Blockchain or return the stored data, emit events, and deploy new contracts at runtime. The semantics of defining smart contracts and available data types vary greatly among the programming languages. For example, in Move[12] language, smart contracts are developed in the form of *modules*. Cadence[13] is a high-level resource-oriented programming language that provides Capability-based Access Control.

The goal of SCIP is to abstract the variety of programming languages in terms of semantics. These data types are directly related to smart contracts and provide a uniform interface for their invocation. Therefore, the features like support for *generics*, object-oriented programming paradigms like inheritance, abstraction, and encapsulation offered by many of these programming languages and their respective platforms are potential features for inclusion into SCIP specification.

Using this criteria, only those platforms should be selected that enable the development of any general-purpose applications and do not restrict users to limited programmability. For example, platforms like Litecoin and Peercoin are programmable but have limited programmability.

Criteria→ **Platform↓**	**Variety**	**Developer friendliness**	**Availability of APIs**	**Use cases**	**Programming language**
Sui [Mys22]	✓	✓	✓	✓	✓
Starknet [Sta21]	✓		✓	✓	✓
Aptos [22]	✓	✓	✓	✓	✓
Cosmwasm [Cos21]	✓		✓	✓	✓
Flow [HSL19]	✓	✓	✓	✓	✓
Cosmos SDK [Ten19]	✓	✓	✓	✓	✓
Polkadot [Web17]	✓		✓	✓	✓
Peercoin [Pee12]	✓	✓	✓		✓ (Limited)
Litecoin [Lit11]	✓	✓	✓		✓ (Limited)
Primecoin [Pri13]	✓				✓ (Limited)
Ethereum Classic [ETC16]		✓	✓	✓	✓
Bitcoin Cash [Bit17]		✓	✓		✓ (Limited)
Bitcoin SV [Bit18]		✓	✓		✓ (Limited)

[12] https://developers.diem.com/docs/technical-papers/move-paper/
[13] https://developers.flow.com/cadence/language

MazaCoin [Tea14]	✓				Unclear
Namecoin [Dur11]	✓	✓	✓		✓ (Limited)
Solana [Sol20]	✓		✓		✓
Arbitrum [Arb20]		✓	✓		✓
Tezos [BB14]	✓	✓			✓
Ploygon [KNAB17]		✓	✓		✓
Optimism [Opt20]		✓	✓		✓
R3 Corda [R316]	✓		✓		✓
Tron [Tro17]		✓	✓		✓
NEAR [NEA18]	✓		✓	✓	✓

Table 4.1: List of blockchain platforms and their suitability to the criteria

4.2 Blockchain analysis

Having laid out the selection criteria for choosing platforms, the subsequent sections present a study of four potential candidate platforms for inclusion into the SCIP gateway and the features that could be part of the SCIP specification. An analysis of these platforms further leads to determining if their SCIP protocol needs modification. The list below mentions properties used for studying the selected platforms and their importance.

- **Network setup**

  When evaluating a blockchain platform, it is important to consider factors such as the consensus mechanism used, the number of validators, the structure of the network, and the level of decentralization. The network infrastructure and scalability should also be considered, as this will determine how well the network can handle a high volume of transactions. Furthermore, the security measures implemented on the network and the ability to recover from a network failure should also be evaluated.

  In summary, understanding the network setup of a blockchain platform is crucial when developing an adapter for the SCIP prototype. Furthermore, the factors related to the network setup should be considered during the implementation of the prototype, as these details need to be abstracted from the client application using the SCIP gateway.

- **Consensus algorithm**

  As discussed in the previous chapter about the importance of the consensus algorithm and its direct correlation to the security and resilience to adversaries, the type of consensus mechanism used is another crucial aspect for the analysis of a blockchain platform. A consensus algorithm also defines the computation power required to operate the network. Therefore, a consensus algorithm of the selected blockchain platform for developing the prototype should also consider environmental sustainability, energy consumption, and costs as a part of its design and implementation. For example, PoS is more energy efficient than the Proof-of-Work consensus algorithm [JHGR20]. Modern platforms like Flow, Sui, Aptos

use the PoS consensus algorithm. Ethereum transitioned to PoS in 2022 because it is more secure, less energy-intensive, and better for implementing new scaling solutions compared to the previous proof-of-work architecture [SHA+].

- **Programming language**

  Section 4.1 discussed about the importance of programming language while selecting a blockchain platform. Even during the analysis phase, studying the platform from the perspective of the programming language it uses will help to check if SCIP needs any changes to allow the programming language features that could be supported in an abstract SCIP specification.

- **Accounts and Security**

  Blockchains work on the principles of cryptographic signatures, making them resistant to tampering and manipulation by malicious actors. However, this does not mean that they are completely immune to security threats. According to a report by SlowMist, over $3.7 billion were lost due to Blockchain related hacks in the year 2022 [Slo23]. The report categorizes the hacks into the following parts: (i) DeFi[14], cross-chain bridges, and NFT (ii) Exchanges (iii) Blockchain (iv) Others . Thus, while considering the security aspect, it is vital to consider what mechanisms are in place to ensure the overall security of the network and deployed applications.

Based on the above mentioned points for analysis of a blockchain platform and the shortlisted platforms from Table 4.1, the further sections present information about Aptos, Sui, Flow, and Cosmos SDK.

4.3 Aptos

Diem, formerly known as Libra, was a digital currency created by Meta (formerly Facebook) to be a low-cost stablecoin that could be used globally. However, it was wound down in 2022 [CFI23]. Later, it led to the announcement of the Aptos blockchain, a spin-off of Deim, from the former employees of Meta. Aptos blockchain is a smart contract platform focusing on the following key principles: scalability, safety, reliability, and upgradeability [22]. In addition, Aptos natively integrates and internally uses the Move language [BCD+20] and PoS algorithm for consensus.

4.3.1 Network setup

The components of Aptos blockchain as as follows:

[14] *Decentralized Finance* (DeFi) refers to the applications providing financial services such as lending, borrowing, and swapping on a blockchain [SK22]

Validator

A validator processes transactions using a BFT proof-of-stake consensus mechanism. Validators are responsible for executing the transactions, validating them and maintaining the integrity of the blockchain by ensuring that only valid transactions are included in new blocks. A validator has to be in the *active* state to participate in consensus. Alternatively, a validator can be *inactive* if it does not have enough stake to participate, rotates out of the validator set, elects to be offline as it synchronizes blockchain state, or is deemed not participating by the consensus protocol due to poor historical performance.

Client

A client is an entity that submits transactions and queries the state and history of the blockchain. In addition, clients can download and verify validator-signed proofs of queried data.

Full node

A full node is a client that replicates the transaction and blockchain state from the validators or other full nodes in the network. In addition, it may elect to prune transaction history and blockchain state to reclaim storage.

Light client

A light client only maintains the current set of validators and can query partial blockchain state securely, typically from full nodes. A wallet is an example of a light client.

Consensus algorithm

Aptos uses DPoS as the consensus algorithm to agree on ordering the blocks and their contents. The validators must have a minimum amount of Aptos token staked, a native currency for the Aptos blockchain. Users can delegate their tokens to validators as a stake and earn rewards proportional to their staked amounts. At the end of every epoch, validators and their respective stakes will receive rewards or get penalized through slashing.

4.3.2 Programming model

In the context of the Aptos ecosystem, a *module* can be mapped to a smart contract that can be invoked, executed, and change the state of the blockchain. Aptos allows clients to submit transactions that can publish new modules, upgrade existing modules, execute *entry functions*[15] defined within a module, or contain scripts that can directly interact with the public interfaces of modules. Module

[15] https://aptos.dev/guides/move-guides/move-on-aptos#visibility

upgradeability differentiates Aptos from Ethereum, where smart contracts are immutable. Aptos ecosystem provides a Move Prover, which enhances security by protecting contract invariants and behaviors.

Move language is inspired by rust programming language. The Move ecosystem mainly contains a virtual machine(VM) compiler. The modules written in Move language are compiled and converted to byte code which the Move VM executes. Move's support for module upgradeability and comprehensive programmability enables seamless configuration changes and upgrades to the Aptos blockchain. Move programming language supports *generics*. In programming, generics refer to a feature that allows the creation of classes, interfaces, and methods that work with different types of data. But, the current SCIP specification does not support generics for invoking smart contracts or querying the state. Thus, creating a window for improving SCIP specification to add support for generics.

Aptos supports the notion of transactions, events, and accounts and has HTTP REST APIs for querying or invoking transactions. During the execution of a transaction, one or more events can be emitted. Each registered event has a unique key that can be used to query event details. Aptos supports only the generation of events during execution and does not allow querying events during transaction execution to enforce the transaction execution to be a function of only current inputs. An account is identified by a unique 256-bit value known as an account address.

A move module is identified by the address of the account where the module is declared, along with a module name. For example, the module identifier *0x1::coin* is deployed by account *0x1* and with name *coin*. The combination of the account address and the module name must be unique. An address owner can publish multiple modules in the form of *package* as a whole on-chain which includes the bytecode and package metadata. A package can be upgradable or immutable and is defined by the package metadata.

4.3.3 Accounts and Security

Hybrid custodial options and flexible key management helps developers to implement a smooth user experience. In addition, Aptos supports shared or autonomous accounts represented entirely on-chain. This allows complex Decentralized Autonomous Organizations (DAOs) to collaboratively share accounts and use these accounts as containers for a heterogeneous collection of resources. Aptos maintains the ledger state by maintaining the state of the account.

4.4 SUI

Sui is a Layer 1 Section 2.1.6 blockchain platform with smart contract capabilities developed by Mysten Labs, primarily focusing on high-speed but low-cost transaction execution. The project, initiated by former Meta employees, is also a spin-off of Deim blockchain. Although Aptos and Sui are both derived from Deim, they differ from each other in their internal working, the Move programming language used in Sui has been modified to suit its programming model[16]. So,

[16] https://www.sotatek.com/aptos-vs-sui-a-fight-of-the-new-generation-layer-1-blockchain-platforms/

considering these factors, the Variety of platforms criteria is not violated. Sui platform supports smart contract execution developed in Sui Move programming language and is secured by a set of *validators*. With an emphasis on horizontal scaling, Sui processes transactions in parallel and thus increases throughput and uses computing resources better. Sui itself is developed in Rust programming language. In addition, Sui platform has SUI token as a native asset that is used for paying transaction fees and also for staking in the delegated PoS algorithm.

4.4.1 Network setup

Sui uses DPoS as a consensus algorithm and has a set of validators executing the transactions in parallel using Byzantine Consistent Broadcast [Wik22] Nodes in Sui blockchain can be a light client or a full node (discussed in Chapter 2).

4.4.2 Consensus algorithm

Sui uses DPoS (described in Chapter 2).

4.4.3 Programming model

Sui Move is derived from the Move programming language. Sui is a distributed ledger that stores programmable objects, each referable by a unique identifier. The state constitutes a pool of programmable objects managed by the Move packages. A move package is a collection of move modules. A move module is analogous to the concepts of *smart contracts* made up of move functions and types. A module can also invoke functions from other move modules. Sui enforces support for type arguments, emitting events and defining custom data types.

4.4.4 Accounts and Security

In Sui, a transaction is valid only if signed using the EdDSA algorithm [Ham15] by the account's private key. If the account does not exist in the blockchain, it is created whenever any asset is transferred. For a transaction to be finalized, it must be submitted to all the validators, has to be certified by the validators, and the certificate has to be shared with all the validators. Sui ensures that the transaction is finalized even when some validators do not act as per the protocol by using cryptographic Byzantine fault tolerant agreement. Certain Sui tokens must be paid as transaction fees even if the transaction is reverted due to an error. This increases the cost of a denial-of-service attack where adversaries try to flood the network with transactions leading to transactions of other users not being finalized. The validators have to stake Sui tokens to be able to participate in the consensus process. Any deviation in the behavior of a validator from the defined protocol leads to slashing, i.e., loss of the stake asset.

4.5 Flow

CryptoKitties is a virtual cat collecting and breeding game built on the Ethereum blockchain, created by Dapper Labs. Players can buy, collect, and sell unique digital cats within the game. In December 2017, the Ethereum network experienced congestion due to the high demand for the blockchain game CryptoKitties [JL21]. The low throughput of Ethereum blockchain led the developers of CryptoKitties to create the Flow blockchain. Flow aims to provide a scalable distributed computing and execution platform for NFT-related applications such as marketplaces and crypto-infused video games. Flow blockchain supports a hybrid set of nodes to reduce congestion and achieve speed and high throughput. It uses PoS as the consensus algorithm. It is a programmable blockchain and uses Cadence programming language for smart contract development. The further content is presented from the Flow's three technical papers [17]:

(i) Technical Paper 1: Separating Consensus & compute [HSL19], (ii) Technical Paper 2: Block Formation [HHS+20], (iii) Technical Paper 3: Execution Verification [HSLZ19], .

4.5.1 Network setup

Flow blockchain defines the following types of nodes to segregate the roles: Collector Nodes, Consensus Nodes, Execution Nodes, and Verification Nodes.

Collector node

Collection nodes are partitioned into approximately equal-sized groups called *clusters*. The transactions received by each cluster are grouped into *collections*. All the collector nodes in the cluster collaborate with each other to generate a collection. The finalized collection generated through consensus among the collectors is called *guaranteed collection*. The hash reference of the collection is then submitted to the consensus nodes for inclusion in a block.

Consensus node

The consensus node maintains the state of the blockchain and appends new blocks. Collector nodes run BFT consensus algorithm to decide which set of received guaranteed collections should be included in the next block. After undergoing through BFT consensus algorithm as *finalized block* consisting of ordered collections is generated. Consensus nodes are also responsible for slashing any non-complying nodes.

[17] https://flow.com/technical-paper

Execution node

Execution nodes execute the set of transactions included in the finalized block. They generate a *execution receipt*. Execution nodes also provide the required information to the verification nodes for examining the result of the execution phase. Execution nodes require high computational resources.

Verification node

A verification node acts as a moderator in the network who provides approval by re-computing the results of the execution node. The verification process is executed in parallel by breaking the result of execution nodes into chunks and examining each chunk independently.

4.5.2 Consensus algorithm

As Flow blockchain uses PoS as a consensus algorithm, the protocol requires that any participant node should stake a certain amount of asset as collateral. The flow ecosystem is open for anyone to participate, so any node with enough stakes can detect any misbehavior by any other node. Upon noticing such incidents, a slashing challenge is initiated, and consensus nodes decide whether the node has committed any non-compliant action. The protocol assumes that 2/3 of the stake is owned by the honest nodes in the network for each type of node.

4.5.3 Programming model

One of the key features of the smart contract programming model in Flow is its use of a new programming language called Cadence. Cadence is a statically typed, object-oriented language with a design focus on safety and security, auditiability, and simplicity. Cadence provides a familiar syntax like Swift, Kotlin, and TypeScript programming language. In addition, it has rich support for data structures, advanced control flow, and capabilities-based access control. Another important feature of Flow's smart contract programming model is its support for composability. This allows developers to create and use reusable smart contract modules that can be combined to perform more complex tasks. Flow also provides a built-in access control mechanism for smart contracts, which allows for fine-grained control over who can access and execute a specific smart contract. This helps ensure that only authorized parties can interact with a contract, which can improve the system's overall security.

Cadence enables resource-oriented programming on Flow. A resource can be a smart contract, digital assets, or balances that are stored in the users' accounts rather than in the form of a data entry in a smart contract in Solidity. Resources can be *created*, *moved*, or *destroyed* by the users who have appropriate access rights. Using Cadence, developers can emit events that can be used by external applications to monitor state updates. In Flow, smart contracts functions can be invoked using a script that is a part of a transaction rather than invoking some function directly with user inputs. A transaction can be signed by multiple users, import multiple smart contracts, and invoke their

functions along with other arbitrary calculations. Flow provides HTTP REST APIs and even client libraries in Python, JavaScript, Swift, and Go to send the transactions, search for events, and query the state. Overall, Flow blockchain is a suitable platform for adding support to the SCIP gateway.

4.5.4 Accounts and Security

Accounts in Flow are composed of the address, balance, public keys, code, and storage. The account addresses in Flow are assigned by an on-chain function that determines the addresses through deterministic sequencing rather than deriving from cryptographic public keys. This allows using the same public key to control multiple accounts or multiple public keys to control a single account. Unlike Ethereum, where an account need not be created explicitly, in Flow, a transaction is required to create an account. While creating an account following information has to be provided: ID (used to identify key within an account), raw public key, signature algorithm, hash algorithm, and weight (integer between 1 and 1000). The Table 4.2 shows the available signature algorithms at the time of writing this thesis. A transaction in Flow is not authorized to access an account unless it has signatures from the accounts having aggregated sum of weight meeting the minimum required threshold of 1000. Signing a transaction in Flow can be a multi-step process. A transaction consists of multiple entities signing it with different purposes: proposer, payer, and authorizers.

Algorithm	**Curve**
ECDSA	P-256
ECDSA	secp256k

Table 4.2: Flow signature algorithm

4.6 Cosmos SDK

Cosmos SDK is an open-source framework for building custom blockchains. Using the cosmos SDK framework, customized, modular, inter-operable blockchains can be built to cater to the application-specific requirements. Developers can either choose PoS or Proof-of-Authority as the consensus algorithm. The framework consists of composable modules which can be tweaked as needed. The framework is written in the Go programming language. Cosmos SDK provides the flexibility to define the application's state, transaction types, and state transition functions by modifying the default setup and replacing any part of it with custom.

4.6.1 Network setup

The blockchain created using Cosmos SDK can be designed to have any desired setup. If the default setup is used without any modification to the consensus algorithm, each node acts as a full node identical to any other node in the network. The network would then consist of a set of validators that run a consensus process and add new blocks.

4.6.2 Consensus algorithm

Cosmos SDK uses Tendermint Core [Int15] as its default consensus engine. Tendermint core is software replicating the state across multiple computing nodes and ensuring resilience to failure even if 1/3 of the nodes fail to operate as per protocol specification. It is a language and application-agnostic component responsible for ordering the transaction bytes and replicating the data. Tendermint splits the consensus process into *rounds*, which is analogous to *epochs* Chapter 2. Validators also vote on the new proposed blocks. There are two voting stages: *pre-vote* and *pre-commit*. A block is added to the blockchain when at least two-thirds of the nodes pre-commit for the same block in the same round. Every pre-commit must be justified by a *polka* in the same round, which occurs when two-thirds of the nodes pre-vote a block. If any participating node deviates from the protocol specification, the staked assets are destroyed through slashing.

4.6.3 Programming model

Cosmos SDK is a framework written in Golang for building custom blockchains. The underlying consensus protocol, i.e., Tendermint, does not impose any programming language-specific constraints on the type of data being added to the blockchain. An application communicates with Tendermint using Application BlockChain Interface (ABCI) as shown in Figure 4.1. A blockchain built using Cosmos can cater to a specific application or be designed to act as a platform to execute smart contracts written using a programming language it supports.

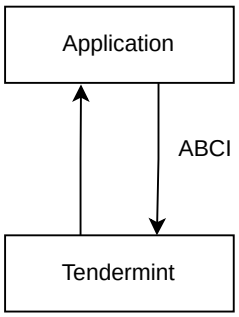

Figure 4.1: Application and Tendermint communicate using ABCI

Cosmos SDK consists of *modules* containing some specialized business logic written in Go programming language. The default setup of Cosmos SDK provides bare minimum modules such as staking, accounts, and token management that are required to build a blockchain. A module handles the message received in the transaction, updates the state, and persists data in a store.

Cosmwasm is a smart contract platform that supports the execution of smart contracts written in Rust[18] programming language. Cosmwasm is a module that can be plugged into any blockchain built on Cosmos SDK to add support for smart contracts.

[18] https://www.rust-lang.org/

4.6.4 Accounts and Security

Developers using Cosmos SDK to build their blockchains can choose the programming language and libraries dealing with the cryptography of their choice. So, the security model of the application and the underlying blockchain depends on the developers rather than Cosmos SDK itself. The default `x/auth` module[19] deals with account types of the application and specifies the base transaction. It handles transaction validation.

4.7 Feature updates to SCIP

From the analysis of the selected blockchains, the following feature updates have been proposed in the SCIP protocol:

- Adding support for generics

  Programming languages like Move and Java provide support for generics and templating. Also, generics might be supported in Solidity programming language in the future[20]. However, the current SCIP specification does not allow calling functions that need type arguments; thus, it does not allow calling transactions for Sui, Aptos, and other blockchains that need type as an argument. Adding support for generics will solve this problem.

- Enabling invocation of transactions with multi-signature support

  Blockchains like Flow and Sui allow calling transactions with multiple signers or accounts/addresses that are controlled through other multiple accounts. Adding support for a multi-signature flow in the SCIP will help abstract these details of handling such transactions from the users and enable collaboration among SCIP clients.

- Adding support for canceling invocation

  When dealing with transactions requiring approvals from multiple clients, adding a feature to cancel the pending invocations or transactions not confirmed on the blockchain will allow clients to build business workflows requiring canceling transactions. The current SCIP specification lacks any feature allowing clients to revoke their transactions that are not confirmed on the blockchain.

- Adding support for replacing invocation

  Similar to adding support for revoking pending invocations, SCIP lacks support for replacing invocations not confirmed on the blockchain. When dealing with transactions requiring approvals from multiple clients, adding a feature to replace the pending invocations or the transactions not confirmed on the blockchain will allow clients to build business workflows requiring replacing them.

The details about adding the above features into the SCIP protocol have been discussed in Chapter 5.

[19] https://docs.cosmos.network/v0.47/modules/auth
[20] https://github.com/ethereum/solidity/issues/869

5 SCIP 2.0

Based on the analysis of blockchain platforms in Chapter 4, the extension to the existing SCIP (referred to as SCIP 1.0 henceforth) specification is discussed in the following sections to accommodate the new features. The SCIP 1.0 introduced a set of *methods*. These blockchain-external consumers are referred to as *client applications* [FBD+20]. The SCIP 1.0 specification proposed 4 *methods* namely: `Invoke`, `Subscribe`, `Unsubscribe` and `Query`. The new SCIP specification (referred to as SCIP 2.0) not only extends these methods but also provides additional methods to utilize new features abstractly and generically without users having to know the internal details of the underlying blockchain technology being used. The methods are namely: `Replace`, `Cancel`, `Sign`, `Get`. All methods return a synchronous response indicating the success or failure of the request message. Some of the methods additionally provide asynchronous responses in the form of callbacks. Method calls require input parameters and return responses composed of *fields*. These fields are mentioned in Section 5.1. The entity providing concrete implementation of the methods that mediates between client applications and multiple heterogeneous blockchains is referred to as *Gateway*. Gateway is reachable using a SCL discussed in Section 2.2.2.

5.1 SCIP 2.0 fields

Table 5.1 specifies the fields and their value in the SCIP protocol which are used with the methods. Newly added fields are highlighted in bold.

Name	Type	Description
Function Identifier	string	the name of the function
Event Identifier	string	the name of the event
Inputs	Parameter[ ]	a list of function inputs
Outputs	Parameter[ ]	a list of function/event outputs
Callback URL	string	the URL to which the callback message must be sent
Correlation identifier	string	a client-provided correlation identifier
Degree of confidence	number	the degree of confidence required from the transaction
Timeout	number	the number of seconds the gateway should wait for the transaction to gain the required degree of confidence
Signature	string	the client's base 64-encoded signature of the contents of a request message

Timestamp	string	the time at which an event occurrence / function invocation happened.
Filter	string	a C-style Boolean expression to select only certain event occurrences or function invocations
Timeframe	string	the timeframe in which to consider event occurrences / function invocations
Occurrences	Occurrence[]	a list of event occurrences / function invocations
Signers	**string[]**	**The addresses or identifier of the entities that are eligble to provide an approval for the invocation**
Minimum number of signatures	**number**	**the minimum count of the entities from the list of Signers that should approve the invocation prior to submission**
Invocation hash	**string**	**The hash of the fields that uniquely identify an invocation (explained in more detail in Section 5.2.3).**
Type Argument	**string[]**	**a list of types used for specifying the types when invoking methods that support generics.**
Parameter		
Name	string	the name of the parameter
Type	JSON Schema	the abstract blockchain-agnostic type of this parameter
Value	any	the value of this parameter
Occurrence		
Parameters	Parameter[ ]	a list of event / function parameters
Timestamp	see above	

Table 5.1: Description of the fields used in SCIP 2.0 request and response messages.

5.2 SCIP 2.0 methods

This section defines the proposed updates to the existing four methods to the SCIP specification and also described the four new methods: `Get`, `Sign`, `Cancel`, `Replace`.

5.2.1 `Invoke` method

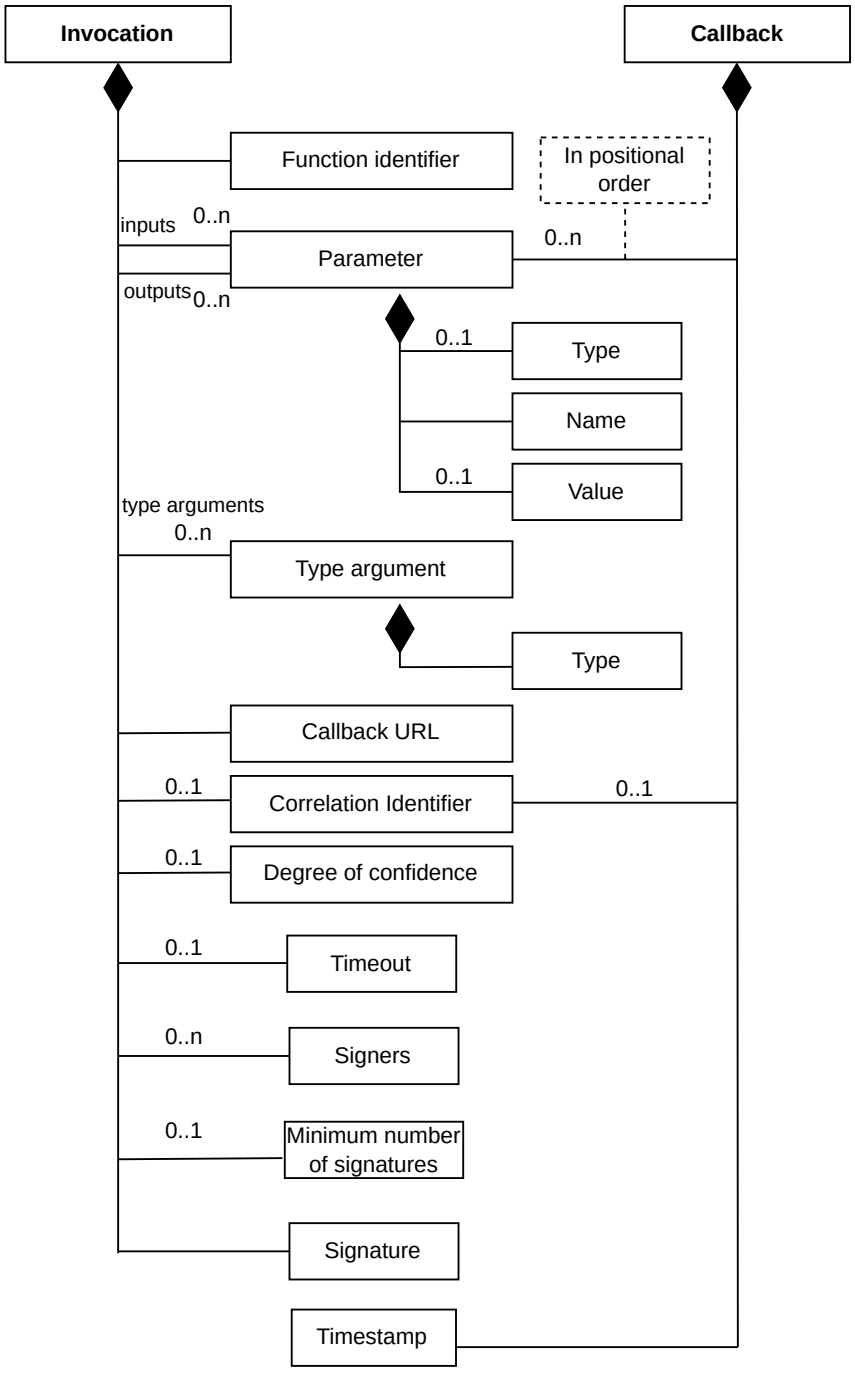

Figure 5.1: The structure of the `Invoke` method with the fields

This method allows external applications to call a smart contract function deployed on a blockchain. The invocation can be gasless if there is no state update in the contract, and the blockchain allows reading the state without paying any fees. Otherwise, the gateway might have to pay certain minimum fees for updating the contract state by sending a certain amount of tokens accepted by the blockchain. For every invocation, the client application must send a *correlation identifier* to identify the invocation request uniquely. In response to the invocation request, the gateway returns a synchronous response indicating whether the request is accepted or rejected. The gateway converts the request into a blockchain transaction specific to the blockchain platform by converting the *Inputs* into its blockchain native request message.

Generics is a feature in many programming languages using which developers can develop functions that run the same logic but on different data types. This powerful feature saves time developing the same business logic on multiple data types. Using the field *type arguments* client application can specify the data types that should be used for the function invocation. This field is optional and only valid when the underlying blockchain platform allows using programming language that supports generics.

As discussed in Chapter 4, each blockchain platform enforces the users to sign the transaction as a security measure to ensure the authenticity of the request. The signing algorithm can differ among the blockchain platforms; hence, to provide a uniform interface to the invocation request, the gateway signs the transaction on behalf of the client application sending the invocation method. However, the gateway itself has a provision to authenticate the validity of the client request by verifying the *Signature* field. Therefore, the client should sign the invocation request using algorithm SSHA256withECDSA-[Cer09].

Assets can be stored on blockchains in standard single-key addresses meaning that whoever has access to the private key can control access to the assets. This means that anyone holding the private key can move the assets from the address without requiring authorization from anybody else. Regarding security, single-key addresses are prone to a single point of failure. If the private key is lost or an unauthorized party gets access to the private key, the assets can be lost forever. To mitigate this risk, multisig wallets can be used to secure access to the funds where the transaction is approved only if a certain number of pre-defined addresses approve a transaction. These types of transactions where approval from multiple user accounts is required can be implemented using a smart contract or can also be a core feature of the blockchain platform e.g., Aptos. The funds in a multisig account can be secured against private key loss or theft. Consider a scenario where a user creates a 2-of-3 multisig account and stores the private keys at different places or devices. Now, even if one private key is lost or stolen, the funds can be accessed using the remaining two accounts. SCIP 1.0 is suitable for single key addresses but lacks the methods to leverage multisig features.

SCIP 2.0 facilitates using multisig account feature by allowing the client application to specify the identifiers (can be a public address or an abstraction over it) of other client applications in the field *Signers*. The field *Minimum number of signatures* can be used to define the threshold of the number of signatures that must be accumulated for the invocation to be executed on the blockchain. The minimum number of signature and signers are optional fields. Suppose these fields are specified by the client application. In that case, the gateway makes the invocation available for discovery to other client application(s) using the **Get** method discussed in Section 5.2.2 and **Sign** method discussed in Section 5.2.3. The value of *Minimum number of signatures* should be less than or equal to the cardinality of the field *Signers*. It is up to the gateway to decide how to accumulate the submitted signatures. After gathering the required approvals from the client applications, the transaction

request is automatically submitted to the blockchain by the gateway. One way the gateway can implement this mechanism is by directly submitting the signatures to the blockchain platform, which enforces multiple-signature verification before transaction execution. Another way is to store the invocation off-chain in permanent storage before submitting the transaction to the blockchain.

A Callback message is sent to the entity listening to the message located at *Callback URL*. The result of the invocation after the request processed by the blockchain platform is sent to the callback URL is sent asynchronously by the gateway. The underlying blockchain either executes the transaction or fails with an error reason. On execution failure, the error message is mapped to one of the errors defined in Section 5.3.

5.2.2 `Get` method

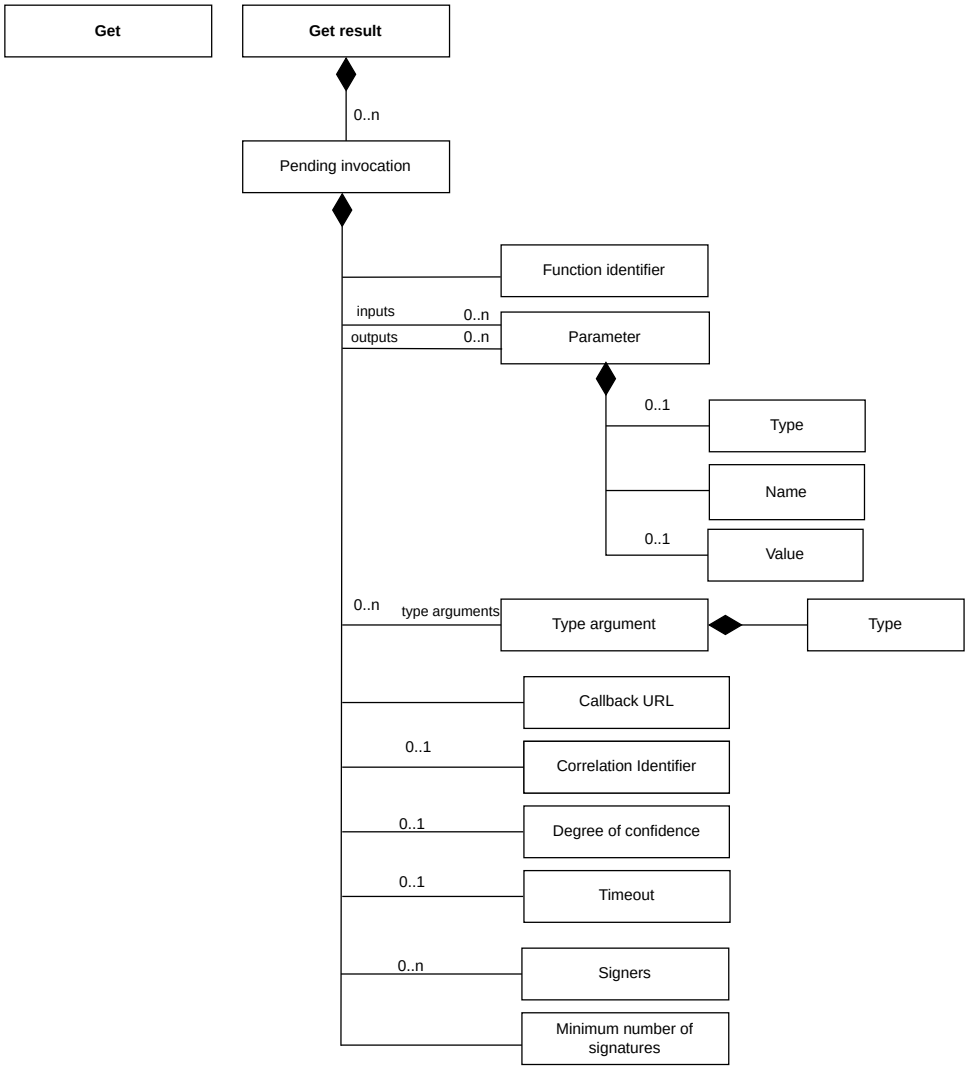

Figure 5.2: The structure of the `Get` method

The `Get` method is accessible to all the client applications so that they can know for which invocations their signatures are required. This read-only method has no state change in the gateway or the blockchain platforms in context. The result is a list of invocations the client applications can examine and provide approval using `Sign` method. `Get` method takes no input parameters. The structure of the response message is explained in Figure 5.2. The gateway provides a synchronous response. The response gives details about the fields and values associated with the invocation. The purpose of providing these details is only to inform the client applications about the invocation parameters.

5.2.3 `Sign` method

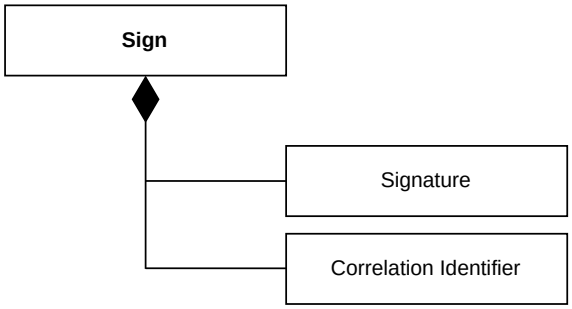

Figure 5.3: The structure of the `Sign`

Using `Sign` method, the client application can provide approval for the transactions proposed by the other client application. This method takes *invocation hash*, and *signature* as input parameters. The structure of the message is explained in Figure 5.3. The client application must sign the *Invocation hash* using ECDSA algorithm.

Invocation hash field is created using the serialized form of the fields: *blockchain identifier*, *smart contract path*, *function identifier*, *inputs*, *outputs*, *type arguments*, *callback URL*, *degree of confidence*, *timeout*, *signers*, *correlation identifier* and *minimum number of signatures* and then hashing it using SHA256 algorithm. *Invocation hash* is required to identify if the fields in the invocation associated with the *correlation identifier* haven't been changed while processing the *Sign* method. Consider a scenario where a user **A** creates an *Invocation* with two signers, namely **B** and **C**. Suppose the *Invocation hash* for this request is **X1** and the client application submits a *Sign* request for this invocation. Now suppose user **A** calls *Replace* with new input parameters for this invocation, and new *Invocation hash* is **X2**. Now suppose, the client application submits a *Sign* request on behalf of the user **C** with invocation hash **X1** and is unaware of the changed invocation fields for the given *correlation identifier*. The gateway should reject this request as the invocation has been updated for the given *correlation identifier*. Any change in the fields related to the invocation request changes the hash, and thus, *Invocation hash* avoids accepting any outdated invocation approvals.

On receiving `Sign` request, the gateway verifies if the public key of the signer is present in the *singers* field for the pending invocation in context and also checks if the signed message corresponds to the latest copy of the *invocation hash*, which the gateway has. If the *invocation hash* does not match

with the version the gateway expects, then the request is rejected with error **SignRejectedError**. After gathering the required valid signatures from the client applications, the SCIP gateway creates a transaction native to the blockchain, submits it, and waits for confirmation from the node, just as the `Invoke` method. Here, the gateway gathers signatures and signs the transaction using an algorithm that the blockchain platform accepts with the corresponding private keys on behalf of the clients, meaning that the gateway is in charge of the private keys used to sign the transaction.

This is a design choice to preserve the essence of abstracting transaction invocation complexities from the clients. An alternative procedure for gathering signatures would be that the gateway generates an unsigned transaction beforehand and provides it to the clients to sign it who hold the private key(s). However, this approach would then force the client to sign a transaction that is specific to a blockchain, which entails that the clients have to understand blockchain-specific transaction structures so that they know what they are signing. The gateway provides a synchronous response with success status if the signature is valid and acceptable for the invocation known by its *correlation identifier*.

5.2.4 `Replace` method

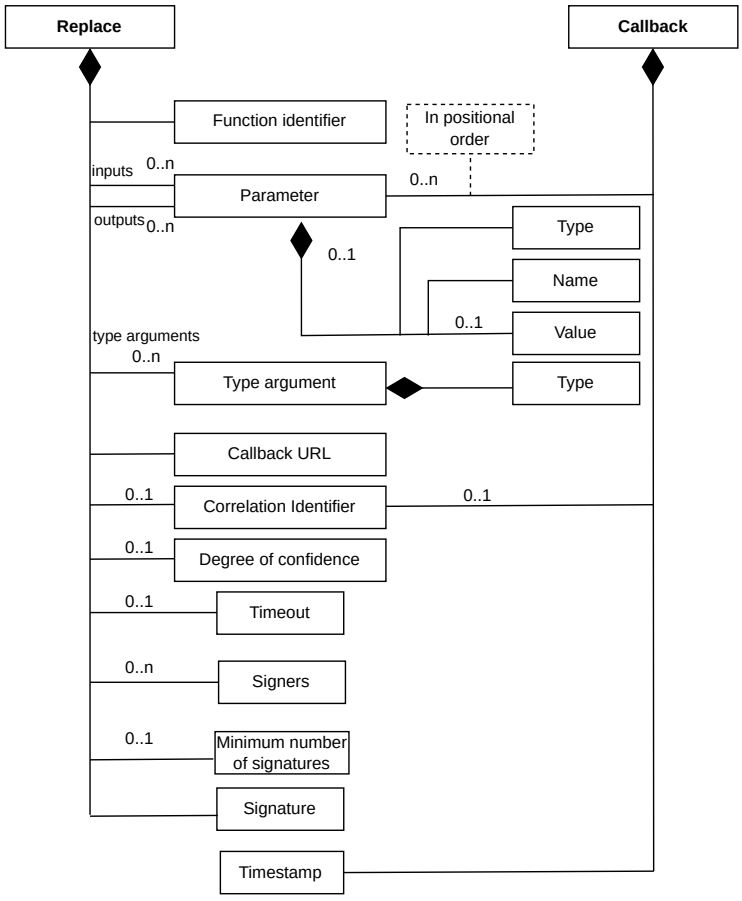

Figure 5.4: The structure of the `Replace` method

As per SCIP 1.0 specification, it is impossible to make changes to an invocation request once it is created and submitted to the gateway. Using `Replace` method specified in SCIP 2.0, the client application can try to make changes to the invocation request provided that the initiator of the request is same client calling this method. For example, suppose the transaction is not yet submitted to the blockchain due to the number of approvals being less than the *minimum number of signatures* and the client sends `Replace` request. In that case, the gateway will replace it with a new invocation, and all the previous approvals will be cleared. All client applications will have to provide new approvals. The *correlation identifier* is used to identify the transaction that should be replaced. Suppose the invocation request is already submitted to the blockchain platform but not yet confirmed. In that case, the gateway submits a new transaction request with the same sequence number as the previous transaction but with a higher gas fee so that the probability of a node picking the new transaction increases. If the transaction is already confirmed, the gateway returns **ReplaceRejectedError** error. The Figure 5.4 mentions all the required parameters to replace an invocation.

5.2.5 `Cancel` method

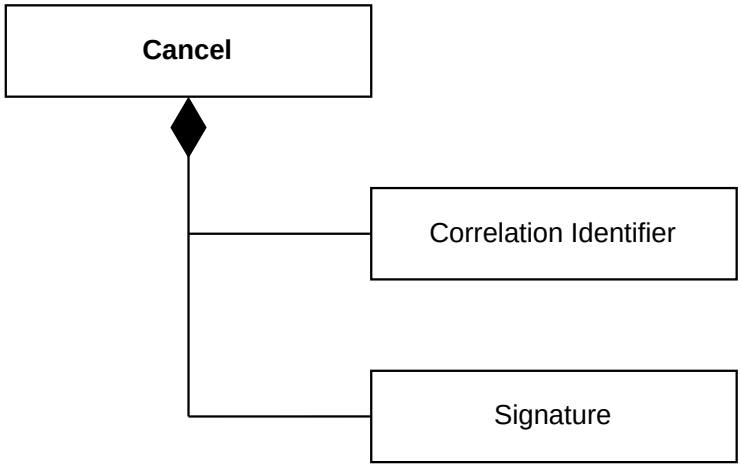

Figure 5.5: The structure of the `Cancel` method

As per SCIP 1.0 specification, it is not explicitly possible to cancel an invocation request once it is created and submitted to the gateway. However, the client application can request the gateway to cancel only its own invocation request using the `Cancel` method specified in SCIP 2.0. The *correlation identifier* is used to identify the transaction that should be reneged and the *Signature* is the signature of the signed invocation hash generated from the fields of the invocation request (described in Section 5.2.3). For example, suppose the transaction is not yet submitted to the blockchain due to the number of approvals being less than the required approval and the client calls the `Cancel` method. In that case, the gateway will remove the invocation request from its own state.

On the other hand, suppose the invocation request is already submitted to the blockchain platform but not yet added to the state of the blockchain. For example, Atpos has a *mempool*[1], which is a list of pending transactions. In that case, the gateway will submit a new transaction request with the same sequence number Chapter 2 as the previous transaction that has no effect on the state of the smart contract in consideration but with a higher gas fee so that the probability of a node picking the new transaction increases. If the transaction is already confirmed, the gateway returns **CancelRejectedError** error.

5.2.6 `Query` method

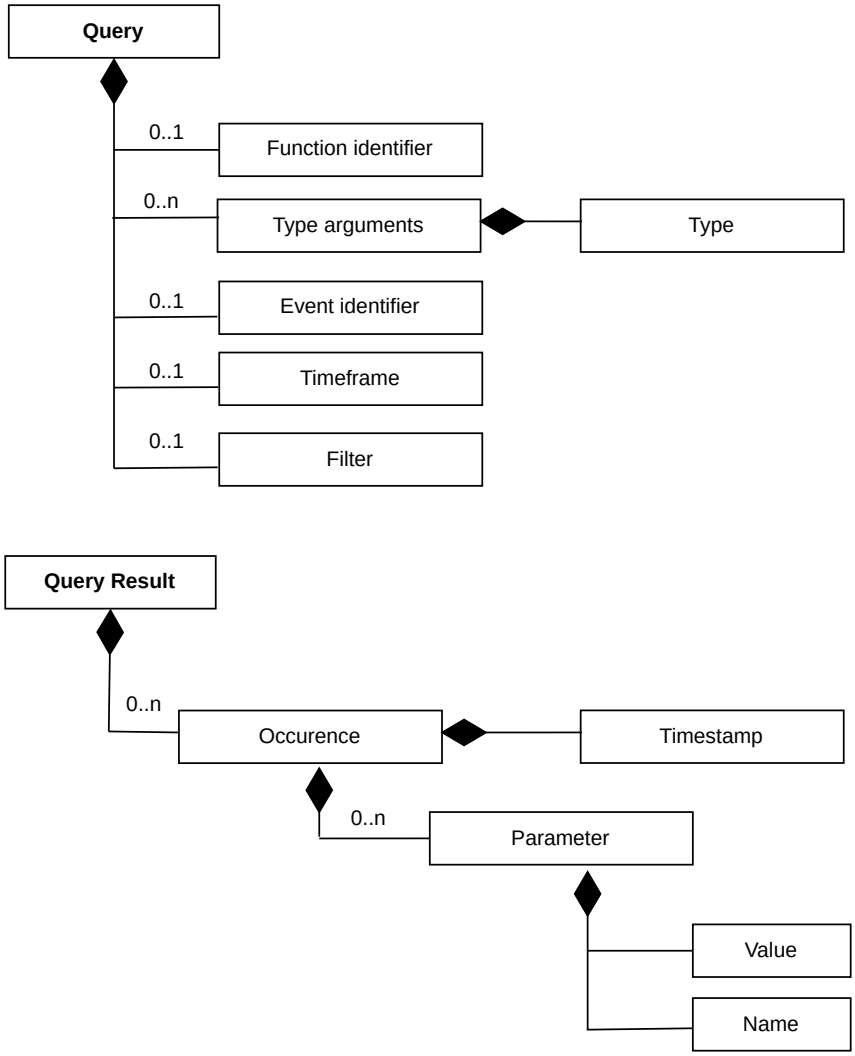

Figure 5.6: The structure of the `Query` method and the result

[1] https://aptos.dev/guides/basics-life-of-txn/

This method allows the client applications to query the invocation of previous events or functions. The query structure and the result message are explained in the Figure 5.6. The *timeframe* field is used to narrow down the scope of the search into a fixed time range period. Suppose the start of the timeframe is not provided. In that case, the gateway considers the genesis block as the start for the search, and similarly, if the end of the timeframe is not provided, then the gateway uses the latest known block as the end. After receiving the *Query* request, the gateway searches the blockchain history for the given combinations of *event identifier* / *function identifier* and the *type arguments*. In response to the Query request, the gateway sends a synchronous response with the list of *occurrences* indicating the parameters types and values of an invocation.

5.2.7 `Subscribe` method

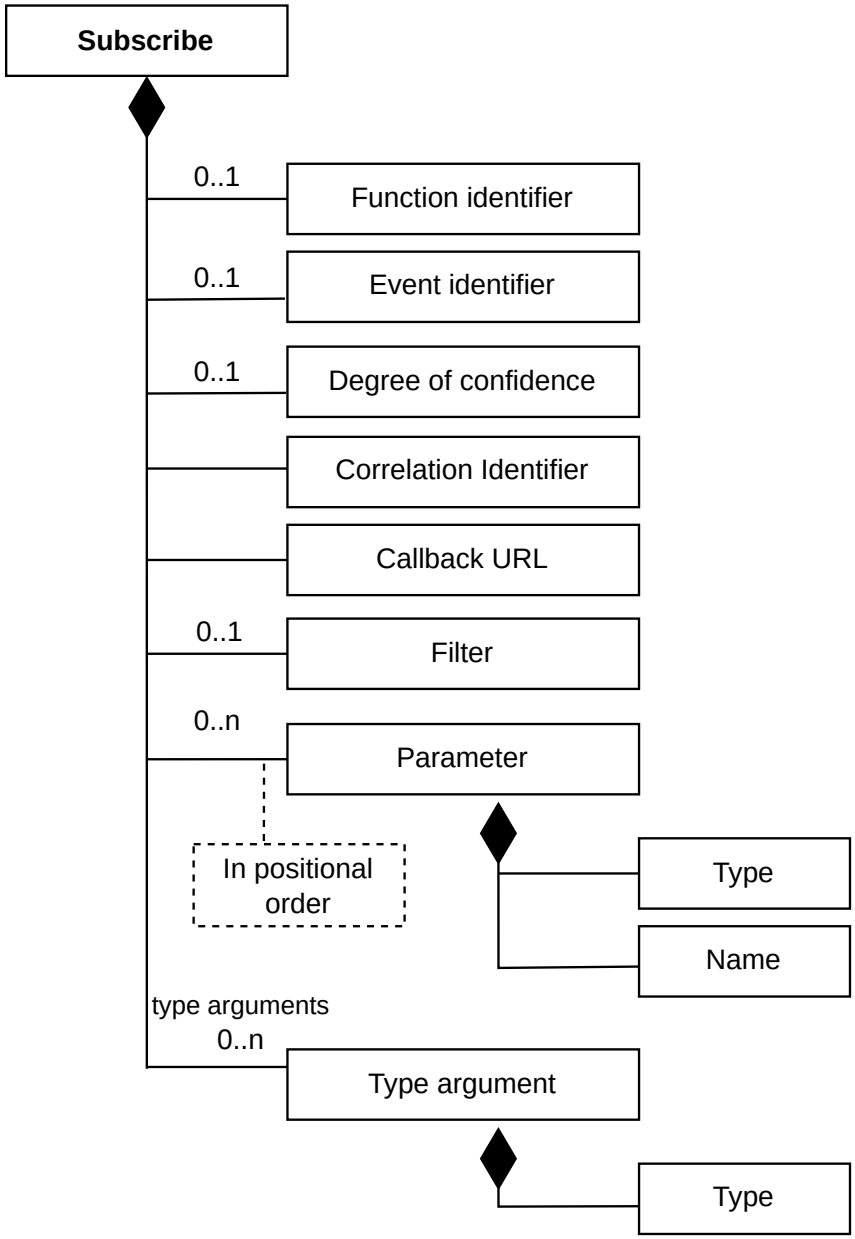

Figure 5.7: The structure of the `Subscribe` method

This method monitors the events/function invocations of a smart contract. The client application can request the gateway to send the information about the invocation as they happen. The structure of the *Subscribe* method is explained in the in Figure 5.7. On receiving the Subscribe request, the gateway sends a synchronous response, whether the request is accepted or rejected, and starts monitoring the smart contract on the blockchain for the event emissions or the function invocation. Additionally, the *filter* parameter can be used to skip the invocations which are not of interest.

Whenever the gateway detects an invocation of the smart contract which matches the criteria of the subscription, the gateway sends the parameters of the invocation along with the *correlation identifier* to the client application listening to the requests at the specified *Callback URL* field in the subscription message. The format of the message sent by the gateway to the client is shown in Figure 5.8. If the client application sends another request subscription request with identical *correlation identifier*, the old request is replaced with a new one.

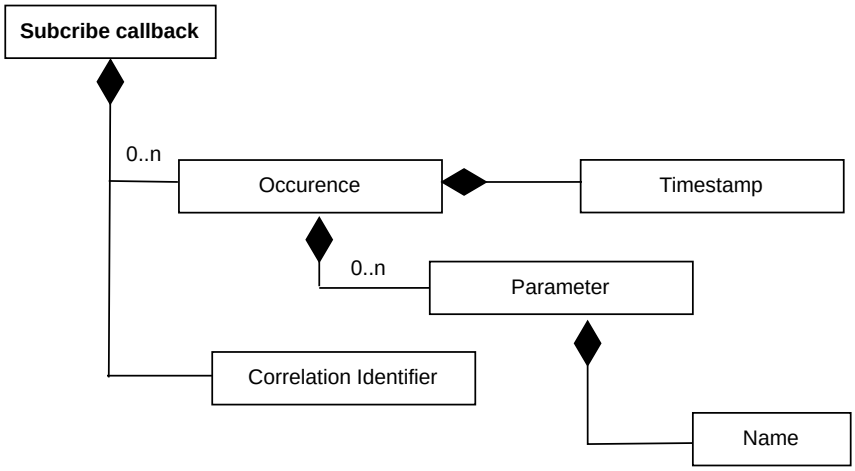

Figure 5.8: Callback message sent from gateway to client when function/event invocation on blockchain matches the subscription parameters

5.2.8 `Unsubscribe` method

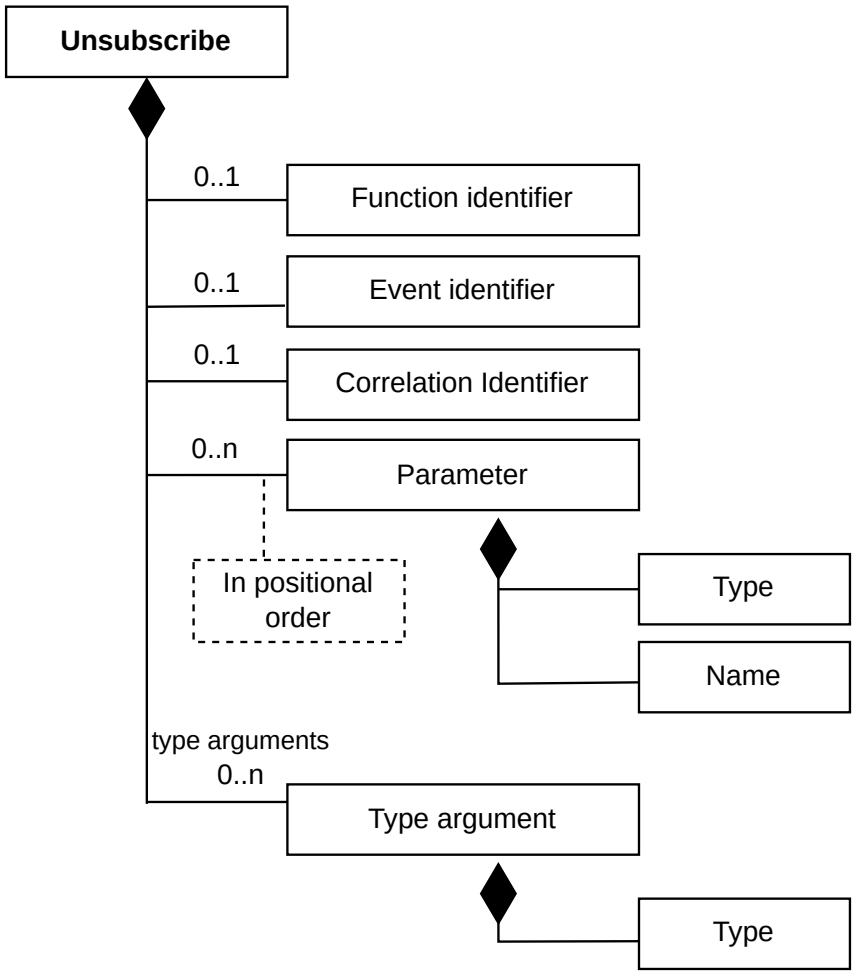

Figure 5.9: The structure of the `Unsubscribe` method

This method is used to cancel the monitoring of the events/function invocations of a smart contract created using the previous *Subscribe* method. The structure of the *Unsubscribe* method is explained in Figure 5.9. On receiving the Unsubscribe request, the gateway sends a synchronous response, whether the request is accepted or rejected, and immediately stops monitoring the smart contract on the blockchain for the event emissions or the function invocation. Using the 4 optional fields, the *Unsubscribe* method provides three valid combinations: (i) if the *correlation identifier* is provided, then only the subscription that corresponds to this id is canceled, (ii) if either *function identifier* or *event identifier* and the *parameters* are provided, then all the individual subscriptions with the these combinations for the mentioned smart contract are canceled, (iii) if no parameters are given, then all the subscriptions of the smart contract are canceled.

5.3 Invocation Errors

SCIP specifies asynchronous and synchronous responses for the request messages that represent successful execution or an error. The error types are split into two types accordingly as described in Table 5.2 and Table 5.3. The new errors added to the possible error types are: **CancelRejectedError**, **ReplaceRejectedError**, and **SignRejectedError**.

Synchronous error	Code	Description
NotFound	-32000	The blockchain instance, smart contract, event or function are not found
InvalidParameters	-32001	Input parameter types, names, or order mismatch the designated function or event. This also indicates inability to map a parameter's abstract type to a native type.
MissingCertificate	-32002	Client certificate is missing
NotAuthorized	-32003	The client application is not authorized to perform the requested task Gateway-side authorization.
NotSupported	-32004	The specified blockchain instance does not support the requested operation.
ConnectionException	-32005	Connection to the underlying blockchain node is not possible.
InvalidScipParam	-32007	A scip method parameter has an invalid value
BalNotAuthorized	-32103	The BAL instance is not authorized to perform the requested operation on the underlying blockchain.
CorrelationIdAlreadyinUse	-32208	The client tried to submit a transaction request with a correlation identifier which is already in use.
CancelRejectedError	**-32209**	**Thrown when the client application requested to cancel an invocation but it already confirmed on the blockchain or signer's public key is not equal to public key of the initiator.**
ReplaceRejectedError	**-32210**	**Thrown when the client application requested to replace an invocation but it already confirmed on the blockchain or signer's public key is not equal to public key of the initiator.**
SignRejectedError	**-32211**	**Thrown when the client application provided approval for an outdated invocation or invocation hash is invalid.**

Table 5.2: Description of synchronous errors sent by the gateway to the client in case of failure to call a method

Asynchronous error	Code	Description
TransactionInvalidatedException	-32006	The transaction associated with an function invocation is invalidated after it was mined.
InvocationError	-32100	A general error occurred when trying to invoke a smart contract function. This error is used when the specific cause of the error cannot be determined.
ExecutionError	-32101	The smart contract function threw an exception
InsufficientFunds	-32102	Not enough funds to invoke the state-changing smart contract function.
Timeout	-32201	Timeout is reached before fulfilling the desired degree of confidence.

Table 5.3: Description of asynchronous errors sent by the gateway to the client in the callback in case of failure to call a method

5.4 Data encoding

In its initial conception, SCIP proposed the use of JSON Schema[2] for hiding the heterogeneity among the data types and values supported by the programming languages used for smart contract development. Based on this work, Table 5.4 shows the mapping between data types that SCIP accepts and the native data types for the specific blockchain. This mapping table is used while developing the plugin prototypes and derived from Cadence values and types[3], SuiJson[4] and Aptos Node API[5].

Json Schema Type	Data type description	Aptos	Sui	Cadence

{
"type ": "integer ",
"minimum ": 0,
"maximum ": 2^M-1
}
	Unsigned integer	u<M>
		max M=256
			u<M>
			max M=256
				UInt<M>
				max M=256

{
"type ": "integer ",
"minimum ": -2^(M-1),
"maximum ": +2^(M-1) - 1
}
	Signed integer	Not supported	Not supported	Int<M>
				max M=256

[2] https://json-schema.org/
[3] https://developers.flow.com/cadence/language/values-and-types
[4] https://docs.sui.io/build/sui-json
[5] https://fullnode.devnet.aptoslabs.com/v1/spec#/

{ "type ": "boolean "}	A boolean value representing 2 possible states: True or False	boolean	boolean	Bool
{ "type ": "string " }	A string type with no length constraint	string	string	String
{ "type ": "string ", "pattern ": "^[0-9a-fA-F]{64}$ " }	A hex encoded 32 byte account address.	address	address	address
{ "type ": "array ", "items ": <type> }	An array of items having uniform data type 'T'	vector<T>	vector<T>	array<T>
{ "type ": "number ", "minimum ": -2^(64-1), "maximum ": +2^(64-1) - 1, "multipleOf ": 10^(-8) }	Signed Fixed-point numbers used for representing fractional values	Not supported	Not supported	Fix64
{ "type ": "number ", "minimum ": 0, "maximum ": 2^64 - 1, "multipleOf ": 10^(-9) }	Unsigned Fixed-point numbers used for representing fractional values	Not supported	Not supported	UFix64

Table 5.4: Mapping between Json schema and native blockchain types for the selected platforms

5.5 JSON RPC Binding

SCIP protocol does not define any specific message communication format or channel. In their paper, Ghareeb Falazi et al. [FBD+20], proposed JSON RPC [JSO10] binding for SCIP. Basedon the proposal, Section 5.5.1, Section 5.5.2, Section 5.5.3 defines the format of requests that a gateway implementing the APIs should accept or respond with.

5.5.1 Request

Listing 5.1 Example: Request message to send a query request

```
{
'jsonrpc': '2.0',
'method': 'Query',
'id': 29433,
'params': {
    'eventIdentifier': 'StringUpdate',
    'filter': '', 'typeArguments': [],
    'timeframe':
        {
            'from': '0',
            'to': '16723413193760000'
        },
    'parameters': []
    }
}
```

5.5.2 Synchronous response

Success

Listing 5.2 Synchronous response body on successful acceptance or execution of request

```
{
    "jsonrpc": "2.0",
    "result": <value>,
    "id": <id>
}
```

Listing 5.3 Example: Response body on successful execution of query request

```
{
"jsonrpc":"2.0",
"id":29433,
"result":
    {
        "occurrences":[
        {
            "parameters":
                [
                    {"name":"oldValue","type":"string","value":"Hello
World!"},
                    {"name":"newValue","type":"string","value":"test
NFT"}
                ],
            "isoTimestamp":"2023-02-06T11:04:51.450Z"
        },
        {
            "parameters":
                [
                    {"name":"oldValue","type":"string","value":"test
NFT"},
                    {"name":"newValue","type":"string","value":"test-2
NFT"}
                ],
            "isoTimestamp":"2023-02-06T11:07:38.299Z"
        }
        ]
    },
}
```

Error

Listing 5.4 Example: Synchronous response body example on error

```
{
"jsonrpc":"2.0",
"id":6472,
"error":{
    "code":-32000,
    "message":"The specified blockchain-id cannot be found"
    }
}
```

5.5.3 Asynchronous response

Success

The asynchronous responses are JSON RPC requests sent from the gateway to the client application.

Listing 5.5 Asynchronous response body for the callback messages sent by the gateway

```
{
    "jsonrpc": "2.0",
    "method": "ReceiveResponse",
    "params": <body>
}
```

Listing 5.6 Example: Asynchronous response body for the callback messages sent by the gateway

```
{
    "jsonrpc":"2.0",
    "method":"ReceiveResponse",
    "params":{
        "correlationIdentifier":"151XT324WV",
        "parameters":[]
    }
}
```

Error

Listing 5.7 Asynchronous response body on error

```
{
    "jsonrpc": "2.0",
    "method": "ReceiveResponse",
    "params":
        {
            "correlationIdentifier": <corelation_identifier>,
            "errorCode": <error_code>,
            "errorMessage": <error_message>
        }
}
```

Listing 5.8 Example: Asynchronous response body on error

```
{
"jsonrpc":"2.0",
"method":"ReceiveResponse",
"params":
        {
            "correlationIdentifier":"EFZ7CWNWOY",
            "errorCode":-32101,
            "errorMessage":"\\"Execution failed.\\""
        }
}
```

6 Prototype

This chapter discusses the background of the SCIP gateway prototype, its architecture, limitations and updates to the prototype to mitigate these limitations in Section 6.1. Later, Section 6.2 explains the changes to the prototype and information on the support for three new blockchain platforms: Sui, Aptos and Flow. Finally Section 6.3, detail out the testing performed to validate the changes to the gateway.

6.1 Background

The Figure 6.1 shows the components of the prototype of the SCIP Gateway before the new methods were introduced. The initial implementation consisted of a JSON-RPC server that accepted the requests from external clients, a *BlockchainManager* that managed the internal state of the application, and *Adapters* wrapped into *Plugins*. The prototype uses a *plugin* based approach to improve the developer workflow for adding support for new blockchains without modifying the other parts of the source code of the prototype. An *adapter* is responsible to pass on the invocation request to the blockchain it has been developed for. An adapter and a blockchain has a 1-to-1 mapping. Due it monolithic nature and design, extending the prototype by adding new adapters required that the developer is aware of the gateway architecture. A *connection profile manager* accepts new configuration values that are required to connect to multiple different instances of the same blockchain type. For example gateway can connect to production instance or a test-net instance of a blockchain type using different blockchain node addresses. The Figure 6.1 shows only SCIP methods in JSON-RPC server and other gateway management APIs are excluded for the purpose of simplicity.

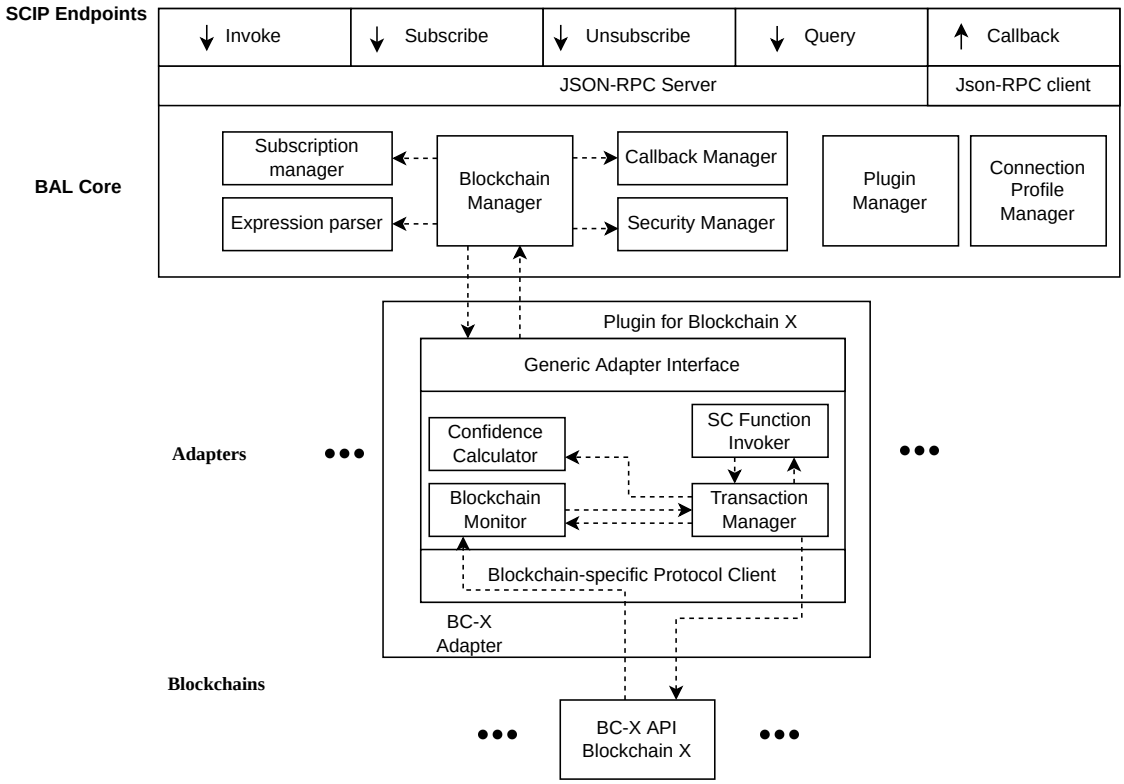

Figure 6.1: Prior architecture of the SCIP gateway prototype before supporting new methods

Figure 6.2 shows the package diagram of the three parts of the prototype (i) Core API, (ii) Application, (iii) Plugin(s) . A brief description of the each of the three parts of the prototype is as follows:

- Core API

  The core API defines interfaces, exceptions, possible transaction states, and common utility methods required to create a plugin. These methods and interfaces are used by the *Application* to invoke some logic on a blockchain but, without having the knowledge of blockchain specific APIs. A *plugin* is responsible for handling the intricacies of interaction with a specific platform.

- Application

  The role of the *Application* component is to offer JSON RPC APIs for to the external client applications and application managers. When an external client application sends any SCIP method request, it finds the appropriate *adapter* instance from pool of available adapters created using active plugins and then routes the message to it. This component also manages the correlation of requests and replies.

- Plugin(s)

For each blockchain type, there is one plugin. The prototype uses PF4J [Sui17] as a plugin framework. Each plugin is responsible for handling the logic to interact with a specific blockchain type and must import the Core API and implement the required interface defined by the Core API. A plugin can be loaded, disabled or even removed at run-time without the need of restarting the whole application. A plugin is *jar* uploaded to the gateway using the plugin management APIs exposed by the application.

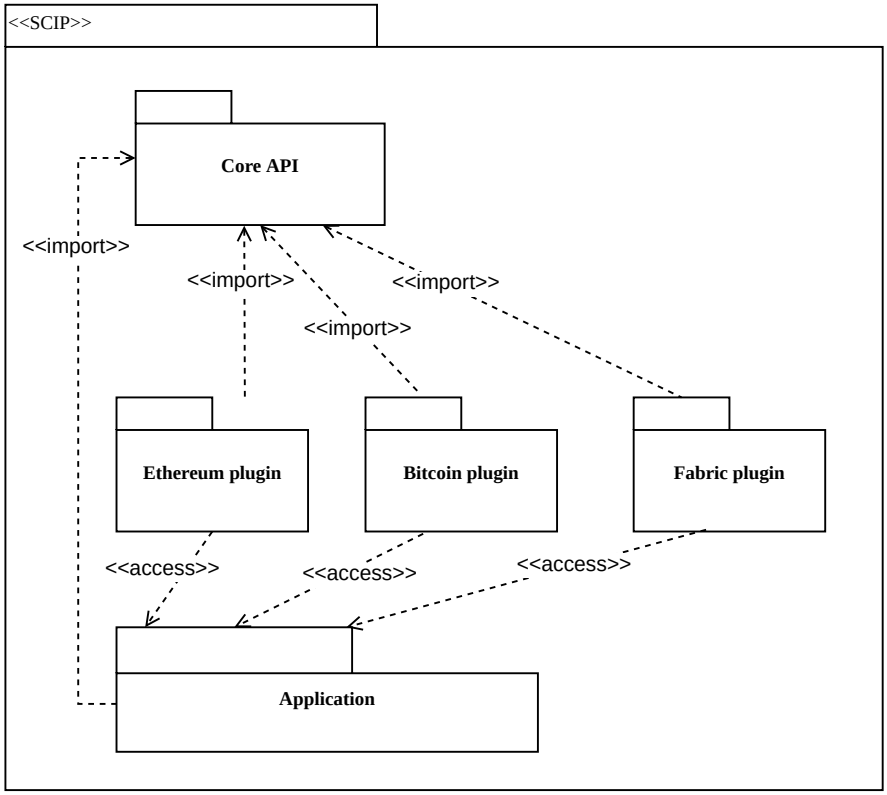

Figure 6.2: Package diagram of SCIP gateway with supported blockchain platforms

Using a plugin based approach allow the developers to create support for new blockchain independently. But, even with this approach, it is required that the private keys are accessible to the gateway for signing the transactions, creating a risk of loss of all the assets that can be controlled using the private keys in case of a security breach. Another drawback of the current gateway framework is that it enforces the developers to develop plugins in Java programming language. Not all blockchain offer a Java client thus forcing developers to re-develop the blockchain specific client using Java. Section 6.2 explains the extension of the current gateway to mitigate the above drawbacks.

6.2 SCIP 2.0 Gateway implementation

This section discusses the updates to the SCIP gateway as per the proposed specification in Chapter 5. The scope of work for the implementation includes extension of the prototype discussed before, support for three new blockchains, implementation of scripts for testing the changes and updates to

the existing plugins. Generic plugin is a separate contribution from the defined scope of work as a part of the thesis. Each of the subsection summarizes the changes in the source code, and mentions the link of the pull request or source code.

Figure 6.3 shows the updated architecture of the prototype. The highlighted components in grey colour are the additions to the gateway implementation. Also, the existing methods have been updated with new fields. The details regarding the updates to the prototype, release versions, links to the Pull request or the repository are discussed further.

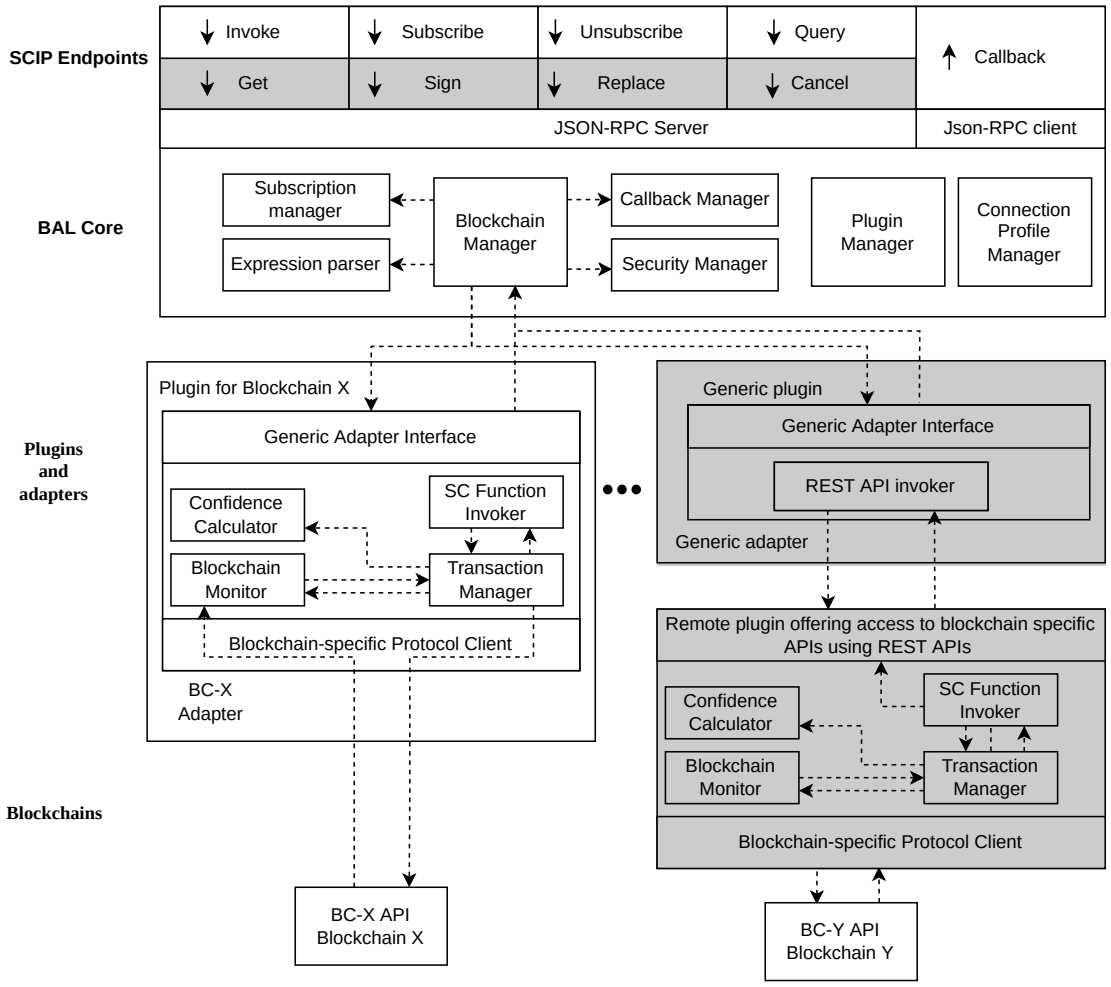

Figure 6.3: SCIP gateway architecture with four new methods and a generic plugin

6.2.1 Core API

Table 6.1 shows the changes to the Core api component. All the plugins that are supported by the updated gateway use the mentioned version.

Version
2.0.7
Pull request
https://github.com/TIHBS/blockchain-access-layer-api/pull/6
Summary of changes
Update interface *BlockchainAdapter.java* to add new proposed fields in existing methods, define new methods that an adapter must implement.
Update interface *BlockchainAdapter.java* to add method delegating the subscription to a service running remotely (discussed further in Section 6.2.4)
Create new error types

Table 6.1: Summary of changes for the Core API in SCIP prototype

6.2.2 Application

Table 6.2 summarizes the changes to the Application component. It includes adding the four new methods introduced in the new proposed SCIP specification. The prior implementation of this component did not include signature verification which is added in the updated version. Also, the application component includes logic to cancel or replace the pending invocations if they have not been submitted to the blockchain. If the invocation request is already submitted to the blockchain, then it calls the plugin's cancel and replace functionality defined in the updated Core API.

Version
2.0.0
Pull request
https://github.com/TIHBS/BlockchainAccessLayer/pull/20
Summary of changes
Update existing SCIP method implementation with new fields
Implement four new SCIP methods as per specification
Implement signature validation logic
Update dependency version of Core api to 2.0.7

Table 6.2: Summary of changes for the Application component in SCIP prototype

6.2.3 Aptos plugin

Aptos provides REST APIs[1] for interacting with its node(s). These REST APIs allow users to send transactions and query the state of the blockchain. Aptos ecosystem does not offer a java client, so a SCIP gateway plugin has been developed using these REST APIs. The REST APIs do not include the endpoints for subscribing to the events and function invocations. So, to overcome this

[1] https://fullnode.devnet.aptoslabs.com/v1/spec#/

limitation and allow the clients using the SCIP gateway to subscribe to the events, the Aptos plugin periodically queries the node using the REST APIs and informs the client by sending callbacks through the gateway. Table 6.3 gives a summary of the plugin offers.

Version
1.0.0
Repository
https://github.com/akshay-ap/bal-aptos-plugin
Supported features
Query event invocations in time frame
Invoke smart contract functions
Subscribe to events
Handle Unsubscription

Table 6.3: Summary of features for the Aptos plugin

6.2.4 Generic plugin

The SCIP gateway prototype is developed in Java programming language which meant the plugins should also be developed using Java programming language before introduction of a *generic plugin*. However, not every blockchain platform has a Java SDK. This creates hindrance in adding new support for such platforms into SCIP. As a solution to let developers focus on integration rather than dealing with implementing the SDK in a particular language, a *generic plugin* is composed of two parts:

- A Java plugin for SCIP gateway which is independent of any blockchain platform. This component calls HTTP REST APIs exposed by some external service for invoking transactions, querying state, and even handling subscriptions.
- An external service which exposes REST APIs for and is aware of handling platform specific requests. The service can be implemented using any programming language thus, providing flexibility for developers to use blockchain SDK available in the language of their choice.

Splitting the working of plugin provides following benefits:

- Developers focus on developing the logic for integration rather than knowing the details of implementing a plugin specifically in Java. The developers must only offer pre-defined REST APIs through any means they choose.
- A remote service can handle SCIP requests that are computation heavy. Thus, a scalable, decentralized SCIP gateway implementation is possible with the existing prototype. Using a remote service, the SCIP gateway implementation can be transformed into a microservice-based architecture where each plugin would be an independent microservice that would offer scalability, greatly eases software maintenance, and impose no additional lock-in [DGL+16]. Developers can freely choose the optimal resources such as languages and frameworks [DGL+16].

- A remote service handles the private keys rather than the gateway, meaning that the responsibility of securely storing and managing the private keys is delegated to the remote plugin located outside the gateway. So, the private is accessible only to the component that requires it rather than the gateway and possibly to other plugins.
- A remote service can be upgraded without restarting the gateway.

The Table 6.4 show the HTTP REST APIs and format of requests that a remote service must offer. As a part of thesis work, two platforms have been integrated into SCIP gateway using generic plugin and the APIs listed below: Flow and Sui. Table 6.5 gives a summary of features supported by the generic plugin.

Invoke method	
Endpoint	<host>/invoke
Method	POST
Description	This method allows the gateway to send Invoke request to a remote service. The request is originated from the client application interacting with the SCIP gateway.
Query parameters	-

Body	Content type: application/json
	Template Example:
	`{`
	`  typeArguments: <array of types>,`
	`  outputs: <array of output parameters>,`
	`  signers: <array of public addresses>,`
	`  smartContractPath: <address of smart contract>,`
	`  inputs: <array of input parameters>,`
	`  requiredConfidence: < float>,`
	`  minimumNumberOfSignatures: <number>,`
	`  functionIdentifier: <function name>,`
	`  timeout: <number>,`
	`  signatures: [{ <public address>: <signature>}, ...]`
	`}`

Response	Template Example:
	`Response code: 200`
	`{ transactionHash : <transaction hash> }`
	`Response code: 4xx`
	`{ errorCode: <scip error code>, errorMessage: <string> }`
	`Response code: 5xx`
	`{ errorCode: <scip error code>, errorMessage: <string> }`

Query method	
Endpoint	<host>/query
Method	POST

Query parameters	-
Description	This method allows the gateway to send Query request to a remote service. The request is originated from the client application interacting with the SCIP gateway.

Body	Content type: application/json
	Template Example:
	
	{
	filter: <string>,
	timeframe: {from: <start time>, to: <end time>},
	smartContractPath: <smart contract path>,
	eventIdentifier: <string>,
	functionIdentifier: <string>
	outputParameters: <list of parameters>,
	inputParameters: <list of parameters>,
	typeArguments: <array of types>
	}

Response	Template Example: Response code: 200 <list of occurrences> Response code: 4xx { errorCode: <scip error code>, errorMessage: <string> } Response code: 5xx { errorCode: <scip error code>, errorMessage: <string> }
Subscribe method	
Endpoint	<host>/subscribe
Method	POST
Query parameters	-
Description	This method allows the gateway to send Subscribe request to a remote service. The remote plugin will start monitoring the blockchain for event/function invocations after validating the request and send occurrences to the client in the form of callbacks.

Body	Content type: application/json
	Template Example:
	
	`{`
	`  smartContractPath: <smart contract path>,`
	`  eventIdentifier: <string>,`
	`  functionIdentifier: <string>,`
	`  degreeOfConfidence: <number>,`
	`  filter: <string>,`
	`  parameters: <list of parameters>,`
	`  callbackUrl: <string>,`
	`  typeArguments: <array of types>,`
	`  correlationId: <string>`
	`}`

Response	Template Example:
	
	`Response code: 200`
	`OK`
	
	`Response code: 4xx`
	`{ errorCode: <scip error code>, errorMessage:`
	`<string> }`
	
	`Response code: 5xx`
	`{ errorCode: <scip error code>, errorMessage:`
	`<string> }`

Unsubscribe method	
Endpoint	<host>/unsubscribe
Method	POST
Query parameters	-
Description	This method allows the gateway to send unsubscribe message to the remote plugin. The remote plugin will stop monitoring the blockchain and remove subscriptions matching the criteria defined in Section 5.2.8

Body	Content type: application/json
	Template Example:
	
	`{`
	`  smartContractPath: <smart contract path>,`
	`  eventIdentifier: <string>,`
	`  functionIdentifier: <string>,`
	`  parameters: <list of parameters>,`
	`  filter: <string>,`
	`  typeArguments: <array of types>,`
	`  correlationId: <string>`
	`}`

Response	Template Example:
	
	`Response code: 200`
	`OK`
	
	`Response code: 4xx`
	`{ errorCode: <scip error code>, errorMessage:`
	`<string> }`
	
	`Response code: 5xx`
	`{ errorCode: <scip error code>, errorMessage:`
	`<string> }`

Cancel method	
Endpoint	<host>/cancel
Method	POST
Description	The gateway calls this API to inform the remote plugin to try to cancel the transaction. The remote plugin checks whether the transaction is still not confirmed. If not confirmed, remote plugin tries to replace this transaction with any another transaction but with same sequence number as the previous transaction.
Query parameters	-

Body	Content type: application/json
	
	Template Example:
	
	`{`
	`  transactionHash : <transaction hash>`
	`}`

Response	Template Example:
	
	`Response code: 200`
	`OK`
	
	`Response code: 4xx`
	`{ errorCode: <scip error code>, errorMessage:`
	`<string> }`
	
	`Response code: 5xx`
	`{ errorCode: <scip error code>, errorMessage:`
	`<string> }`

Replace method	
Endpoint	<host>/replace
Method	POST

Description	The gateway calls this API to inform the remote plugin to try to replace the transaction. The remote plugin checks whether the transaction is still not confirmed. If not confirmed, the remote plugin tries to replace this transaction with a new transaction with given inputs, but with the same sequence number as the previous transaction from the same account that initiated the earlier transaction.
Query parameters	-

Body	Content type: application/json
	
	Template Example:
	`{`
	`  transactionHash : <transaction hash>,`
	`  typeArguments: <array of types>,`
	`  outputs: <array of output parameters>,`
	`  signers: <array of public addresses>,`
	`  smartContractPath: <address of smart contract>,`
	`  inputs: <array of input parameters>,`
	`  requiredConfidence: < float>,`
	`  minimumNumberOfSignatures: <number>,`
	`  functionIdentifier: <function name>,`
	`  timeout: <number>,`
	`  signatures: [{ <public address>: <signature>},`
	`...]`
	`}`

Response	Template Example:
	
	`Response code: 200`
	`OK`
	
	`Response code: 4xx`
	`{ errorCode: <scip error code>, errorMessage:`
	`<string> }`
	
	`Response code: 5xx`
	`{ errorCode: <scip error code>, errorMessage:`
	`<string> }`

Table 6.4: HTTP REST APIs that a remote service implements when using a generic plugin

Version
1.0.0
Repository
https://github.com/TIHBS/bal-generic-plugin
Supported features combined with generic plugin
Query event invocations in time frame by calling remote plugin through REST API
Invoke smart contract functions by calling external service through REST API
Subscribe to events/function invocations by periodically fetching new events/function invocations from an external service using REST API.
Handle Unsubscription
Perform delegated subscription and unsubscription using REST API

Table 6.5: Summary of features for the Generic blockchain plugin

6.2.5 Flow blockchain plugin

Flow ecosystem provides a JavaScript client to interact with its nodes. A plugin for supporting interaction with the Flow blockchain has been added using this Flow Client Library (JS)[2]. Using a generic plugin and Express[3] framework to provide a service that implements the APIs described in Table 6.4, support for a new blockchain has been successfully added to the SCIP gateway. Table 6.6 gives the information about the supported features for the remote Flow plugin.

Version
1.0.0
Repository
https://github.com/TIHBS/bal-flow-plugin
Supported features combined with generic plugin
Query event invocations in time frame
Invoke smart contract functions
Subscribe to events
Handle Unsubscription

Table 6.6: Summary of features for the Flow blockchain plugin

[2] https://developers.flow.com/tools/fcl-js/reference/api
[3] https://expressjs.com/

6.2.6 Sui blockchain plugin

Sui ecosystem provides a TypeScript SDK to interact with its nodes. A plugin for supporting interaction with the Sui blockchain has been added using this Sui TypeScript SDK[4]. Using a generic plugin and Express framework to provide a service that implements the APIs described in Table 6.4, support for a new blockchain has been successfully added to the SCIP gateway. Table 6.7 gives the information about the supported features for the remote Sui plugin.

Version
1.0.0
Repository
https://github.com/TIHBS/bal-sui-plugin
Supported features combined with generic plugin
Query event invocations in time frame
Query function invocations
Invoke smart contract functions
Subscribe to events
Handle Unsubscription

Table 6.7: Summary of features for the Sui blockchain plugin

6.2.7 Ethereum

Table 6.8 provides the information about the changes to the Ethereum plugin for SCIP gateway so that it can be used with the updated gateway implementation.

Version
2.0.0
Pull request
https://github.com/TIHBS/blockchain-access-layer-ethereum-plugin/pull/3
Summary of changes
Update api version to 2.0.7

Table 6.8: Summary of changes for the Ethereum blockchain plugin

6.2.8 Hyperledger Fabric

Table 6.9 provides the information about the changes to the Hyperledger Fabric plugin for SCIP gateway so that it can be used with the updated gateway implementation.

[4] https://github.com/MystenLabs/sui/tree/main/sdk/typescript

Version
2.0.0
Pull request
https://github.com/TIHBS/blockchain-access-layer-fabric-plugin/pull/3
Summary of changes
Update api version to 2.0.7

Table 6.9: Summary of changes for the Hyperledger fabric blockchain plugin

6.3 Testing and Case study implementation

Testing is a critical aspect of software development that ensures the quality and reliability of the code. In software development, various testing methods, including integration testing, test the interactions between multiple system components. The previous SCIP framework lacked integration testing that deployed all the components like different blockchain nodes, SCIP gateway, and plugins. To fill this gap, during the development of the SCIP plugins and validation of changes, a repository dedicated to integration testing that manages multiple plugins and multiple flows for a comprehensive coverage of gateway features has been introduced. The setup process is done using docker-compose so that CI/CD [MFH22] flows can be established. As a part of the testing, Python scripts have been developed which deploy plugins, upload connection profiles, and use SCIP gateway APIs to invoke blockchain transactions and query events. Using the testing framework, developers can reduce the time and effort required for setup and manually ensure that the system works as expected in all scenarios. All the setup instructions and the tests are available at this URL: https://github.com/TIHBS/BAL-Tests. As part of integration tests, each of the individual plugins has unit tests in its respective code repository. The Table 6.10 summarizes the implemented integration tests for each of the plugins.

Plugin → **Test case ↓**	Flow	Sui	Aptos
Invoke transaction with single signer	✓	✓	✓
Invoke transaction with multiple signers	✓	✓	✓
Query function invocation		✓	
Query event invocation	✓	✓	
Query event invocation with filter	✓	✓	
Subscribe to events	✓	✓	✓
Subscribe to function invocation		✓	
Unsubscribe	✓	✓	✓

Table 6.10: SCIP plugin test case implementation

A test case for a multi-signature process shown in Figure 6.4 assuming that all the request execute successfully is as follows:

The first step is for Client 1 to call the invoke method with 1 additional signer. This initiates the multi-signature process, which requires more than one signature to execute a transaction on the blockchain ❶. The gateway stores the state of the pending invocation in an in-memory store ❷ and returns a synchronous acknowledgement after validating the parameters of the `Invoke` request ❸. After the invoke method is called, the Gateway validates the signature to ensure that it is authentic. In the next step, Client 2 fetches the pending invoke using the `Get` method ❹. This allows Client 2 to view the details of the transaction and determine if they want to provide approval for it. If Client 2 decides to approve the transaction, they can do so by using the `Sign` method ❻. This provides the necessary signature count to execute the transaction on the blockchain. After Client 2 provides the approval, the Gateway validates the signature to ensure that it is authentic and returns synchronous response `OK` ❼. If not, appropriate error is returned. Once the signature is validated, the Gateway sends the transaction to the plugin. The plugin then creates a blockchain-specific transaction with the multi-signer account ❽, which allows multiple parties to approve the transaction before it is executed on the blockchain. After broadcasting the request ❾, the gateway monitors the state of the transaction(❿ and ⓫). Finally, after the transaction is confirmed and required block confirmations are acquired, the Gateway provides a callback, which informs the client which initiated the invocation request that the transaction has been executed successfully on the blockchain(⓬).

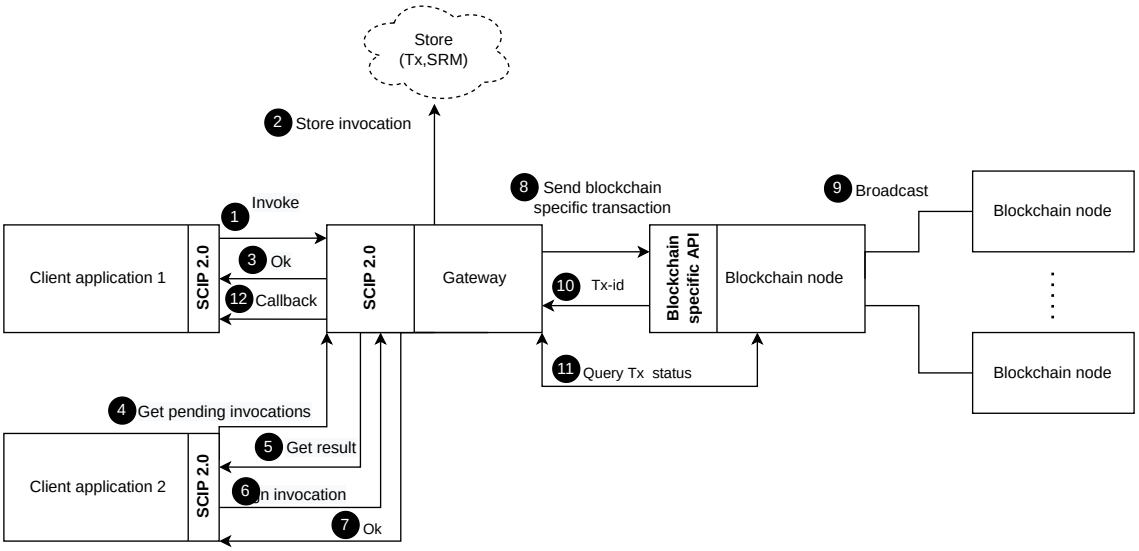

Figure 6.4: Case study implementation showing a workflow of multiple clients signing a transaction

The above mentioned flow has been implemented using the Flow plugin for SCIP gateway. The scripts for the case study are available the URL:`https://github.com/TIHBS/BAL-Tests/tree/main/tests/case_study/flow`. The screenshots of the logs of the case study are shown in Figure 6.5, Figure 6.6, Figure 6.7, Figure 6.8 and Figure 6.8, and Figure 6.9.

```
(venv) python -m unittest tests.case_study.flow.test_client_1.TestClient1.test_initiate_invocation
Pending invocations before initiating Invoke request: 0
Invoke request url: [http://localhost:9091/blockchain-access-layer/webapi?blockchain=generic-plugi
n&blockchain-id=flow-1&address=0xf8d6e0586b0a20c7/Example]
Invoke request body:
{'jsonrpc': '2.0', 'method': 'Invoke', 'id': 3632, 'params': {'functionIdentifier': 'setValues', '
inputs': [{'name': 'name', 'type': '{"type":"string"}', 'value': 'test NFT'}, {'name': 'newBoolean
Var', 'type': '{"type":"boolean"}', 'value': 'true'}, {'name': 'newInt8Var', 'type': '{"type":"int
eger","maximum": "127", "minimum": "-128"}', 'value': '-10'}, {'name': 'newUInt128Var', 'type': '{
"type":"integer","maximum": "340282366920938463463374607431768211455", "minimum": "0"}', 'value':
'1000'}], 'outputs': [], 'timeout': 1000000, 'doc': 0, 'callbackUrl': 'http://localhost:5010', 'si
gnature': 'MEQCICwAmv9Xut8zQi3a57XhXk8cJp1I4fR0gvtDcOwY9REuAiAqYoN+cBojXT1ZVkPJTXyIFE4KCydJJHoMf7W
miUl8LA==', 'proposer': 'MFYwEAYHKoZIzj0CAQYFK4EEAAoDQgAEaFkfNVli0NOECZkFCNij3InuDB1Aqi9fa/JQ70aYm
/t0bNWuOMJCy2q0CgWN7OB0CwzMGyT1R8Y1R8AYMWgjYw==', 'correlationIdentifier': '3YVZIL9VS2', 'typeArgu
ments': [], 'signers': ['MFYwEAYHKoZIzj0CAQYFK4EEAAoDQgAEdAUQgW3mKr4z/vnFVZHYHYLOu4sJgupLx/bcdNcwb
JXKwNLeTx8ObZWDc1vif62wqMkFdc9BxxL/J0CINRU5xQ==', 'minimumNumberOfSignatures': 1}}
Invoke response code: [200]
Invoke response body:
[{'id': 3632, 'result': 'OK', 'jsonrpc': '2.0'}]
Pending invocations after initiating Invoke request: 1
.
----------------------------------------------------------------------
Ran 1 test in 0.238s

OK
(venv)
```

Figure 6.5: Screenshot of the logs from Client-1 initiating the `Invoke` request

```
(venv) python -m unittest tests.case_study.flow.test_client_2.TestClient2.test_sign_invocation
Signing pending invocation:
{'blockchainIdentifier': 'flow-1', 'correlationIdentifier': '3YVZIL9VS2', 'inputs': [{'name': 'nam
e', 'type': '{"type":"string"}', 'value': 'test NFT'}, {'name': 'newBooleanVar', 'type': '{"type":
"boolean"}', 'value': 'true'}, {'name': 'newInt8Var', 'type': '{"type":"integer","maximum": "127",
 "minimum": "-128"}', 'value': '-10'}, {'name': 'newUInt128Var', 'type': '{"type":"integer","maxim
um": "340282366920938463463374607431768211455", "minimum": "0"}', 'value': '1000'}], 'outputs': []
, 'signers': ['MFYwEAYHKoZIzj0CAQYFK4EEAAoDQgAEdAUQgW3mKr4z/vnFVZHYHYLOu4sJgupLx/bcdNcwbJXKwNLeTx8
ObZWDc1vif62wqMkFdc9BxxL/J0CINRU5xQ==', 'typeArguments': [], 'requiredConfidence': 0.0, 'callback
Url': 'http://localhost:5010', 'signature': 'MEQCICwAmv9Xut8zQi3a57XhXk8cJp1I4fR0gvtDcOwY9REuAiAqY
oN+cBojXT1ZVkPJTXyIFE4KCydJJHoMf7WmiUl8LA==', 'proposer': 'MFYwEAYHKoZIzj0CAQYFK4EEAAoDQgAEaFkfNVl
i0NOECZkFCNij3InuDB1Aqi9fa/JQ70aYm/t0bNWuOMJCy2q0CgWN7OB0CwzMGyT1R8Y1R8AYMWgjYw==', 'minimumNumber
OfSignatures': 1, 'signatures': [], 'functionIdentifier': 'setValues', 'smartContractPath': '0xf8d
6e0586b0a20c7/Example', 'timeoutMillis': 1000000, 'invocationHash': 'a3a63d41af3ce84d3440d7c5188e4
5fcbca2a928153cf307bc3f2fc9bb4d117f', 'submitted': False}
Signing invocation with correlation_identifier:[3YVZIL9VS2] using public_key:[MFYwEAYHKoZIzj0CAQYF
K4EEAAoDQgAEdAUQgW3mKr4z/vnFVZHYHYLOu4sJgupLx/bcdNcwbJXKwNLeTx8ObZWDc1vif62wqMkFdc9BxxL/J0CINRU5xQ
==]
Sign response code:
[200]
Sign response body:
[{'id': 1, 'result': True, 'jsonrpc': '2.0'}]
.
----------------------------------------------------------------------
Ran 1 test in 0.073s

OK
(venv)
```

Figure 6.6: Screenshot of the logs from Client-2 signing the request created by Client-1

```
 DEBUG com.github.arteam.simplejsonrpc.server.JsonRpcServer - Request : {"jsonrpc":"2.0","method":"Get","id":1,"params":{}}
 INFO blockchains.iaas.uni.stuttgart.de.jsonrpc.BalService - Get method is executed!
 DEBUG com.github.arteam.simplejsonrpc.server.JsonRpcServer - Response: {"id":1,"result":[],"jsonrpc":"2.0"}
DEBUG com.github.arteam.simplejsonrpc.server.JsonRpcServer - Request : {"jsonrpc":"2.0","method":"Invoke","id":3632,"params":{"function
INFO blockchains.iaas.uni.stuttgart.de.jsonrpc.BalService - Invoke method is executed!
DEBUG com.github.arteam.simplejsonrpc.server.JsonRpcServer - Response: {"id":3632,"result":"OK","jsonrpc":"2.0"}
DEBUG com.github.arteam.simplejsonrpc.server.JsonRpcServer - Request : {"jsonrpc":"2.0","method":"Get","id":1,"params":{}}
INFO blockchains.iaas.uni.stuttgart.de.jsonrpc.BalService - Get method is executed!
DEBUG com.github.arteam.simplejsonrpc.server.JsonRpcServer - Response: {"id":1,"result":[{"blockchainIdentifier":"flow-1","correlationId
DEBUG com.github.arteam.simplejsonrpc.server.JsonRpcServer - Request : {"jsonrpc":"2.0","method":"Get","id":1,"params":{}}
INFO blockchains.iaas.uni.stuttgart.de.jsonrpc.BalService - Get method is executed!
DEBUG com.github.arteam.simplejsonrpc.server.JsonRpcServer - Response: {"id":1,"result":[{"blockchainIdentifier":"flow-1","correlationId
DEBUG com.github.arteam.simplejsonrpc.server.JsonRpcServer - Request : {"jsonrpc":"2.0","method":"Sign","id":1,"params":{"signature":"ME
INFO blockchains.iaas.uni.stuttgart.de.jsonrpc.BalService - Sign method is executed!
```

Figure 6.7: Screenshot of the logs of SCIP Gateway showing requests from clients

Figure 6.8: Screenshot of the logs from the remote plugin for the Flow blockchain receiving the `Invoke` request from the SCIP gateway

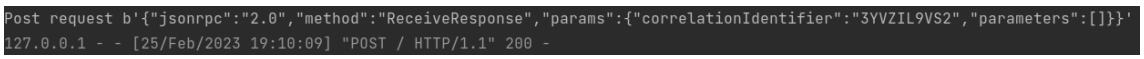

Figure 6.9: Screenshot of the logs from the Callback handler process showing successful execution of the `Invoke` request initiated by the Client-1

In Section 4.1 the research question: "How can blockchains be selected for analysis? (RQ1)" has been addressed by describing the search methodology and five properties for selection: Variety of platforms, Developer friendliness, Availability of APIs, Use cases, and Programming language. Section 4.2 answers the research question:"How to analyze blockchain technology? (RQ2)" by describing four key aspects: Network setup, Consensus algorithm, Programming language, Accounts and Security. Finally, "What updates can be proposed to SCIP for it to be suitable for new blockchains? (RQ3) is addressed in Chapter 5, and the implementation details of the prototype have been described in this chapter and thus, addressing all the research questions of this thesis.

7 Conclusion and Outlook

This thesis work has aimed to address the problem statement of selecting blockchain for studying their properties, analysis of selected blockchain platforms, and selecting features for SCIP protocol by analysis of new blockchain platforms. To address the problem statement, this work first defined blockchains and their terminology, then defined discovery and selection criteria for the blockchain. Through selection, three blockchain technologies and an SDK was shortlisted to achieve the goals in the given timeline. Further, this work described blockchain analysis methodology to understand the selected blockchain platforms for feature selection. The result of the analysis described in the work led to the requirement of improving SCIP without removing any of the previous parts of the specification. During the implementation phase of the work, three new blockchain platforms have been integrated into the SCIP gateway, and a testing framework that was previously missing has been created. A demonstration of setup that simplifies gateway and plugin deployment using the container and virtualization technology like docker would help create a smooth user onboarding process and experience. As an additional outcome of the work, *generic plugin* opens the possibilities of adding other platforms in the future with ease and offers a micro-service-based implementation of the Gateway to manage multiple blockchain support simultaneously. Overall, all the goals set during the initial phase of the thesis work have been completed.

The future outlook could be a whole ecosystem of tools that offer integration services with a minimal entry-level barrier to developers from all skill and experience levels.

Bibliography

[22] *The Aptos Blockchain: Safe, Scalable, and Upgradeable Web3 Infrastructure*. Aug. 2022. URL: https://aptos.dev/assets/files/Aptos-Whitepaper-47099b4b907b432f81fc0effd34f3b6a.pdf (visited on 10/09/2022) (cit. on pp. 40, 42).

[AAA13] A. A. Alkandari, I. F. Al-Shaikhli, M. A. Alahmad. "Cryptographic Hash Function: A High Level View". In: *2013 International Conference on Informatics and Creative Multimedia*. 2013, pp. 128–134. DOI: 10.1109/ICICM.2013.29 (cit. on p. 24).

[ACG15] B. Akins, J. Chapman, J. Gordon. "A Whole New World: Income Tax Considerations of the Bitcoin Economy". In: *Pittsburgh Tax Review* 12 (Feb. 2015). DOI: 10.5195/taxreview.2014.32 (cit. on p. 15).

[Alc22] Alchemy. *Archive Nodes - Everything You Need to Know*. 2022. URL: https://www.alchemy.com/overviews/archive-nodes (cit. on p. 22).

[Arb20] Arbitrum Team. *Arbitrum*. https://arbitrum.io/. 2020 (cit. on p. 41).

[ASZ22] F. E. Alzhrani, K. A. Saeedi, L. Zhao. "A Taxonomy for Characterizing Blockchain Systems". In: *IEEE Access* 10 (2022), pp. 110568–110589. DOI: 10.1109/ACCESS.2022.3214837 (cit. on p. 24).

[BB14] A. Breitman, K. Breitman. *Tezos: A Self-Amending Crypto-Ledger*. https://tezos.com/. 2014 (cit. on p. 41).

[BCC+21] L. Breidenbach, C. Cachin, B. Chan, A. Coventry, S. Ellis, A. Juels, F. Koushanfar, A. Miller, B. Magauran, D. Moroz, S. Nazarov, A. Topliceanu, F. Tramèr, F. Zhang. *Chainlink 2.0: Next Steps in the Evolution of Decentralized Oracle Networks*. 2021. URL: https://research.chain.link/whitepaper-v2.pdf (visited on 01/06/2023) (cit. on p. 35).

[BCD+20] S. Blackshear, E. Cheng, D. L. Dill, V. Gao, B. Maurer, T. Nowacki, A. Pott, S. Qadeer, Rain, D. Russi, S. Sezer, T. Zakian, R. Zhou. *Move: A Language With Programmable Resources*. 2020. URL: https://diem-developers-components.netlify.app/papers/diem-move-a-language-with-programmable-resources/2020-05-26.pdf (cit. on p. 42).

[Bin18] Binance Academy. *What is a Multisig Wallet*. Dec. 2018. URL: https://academy.binance.com/en/articles/what-is-a-multisig-wallet (visited on 01/21/2023) (cit. on p. 30).

[Bit17] Bitcoincash community. *Bitcoin Cash*. https://www.bitcoincash.org/. 2017 (cit. on p. 40).

[Bit18] Bitcoin Association. *Bitcoin SV*. https://bitcoinsv.io/. 2018 (cit. on p. 40).

[BKN+21] M. N. M. Bhutta, A. Khwaja, A. Nadeem Al Hassan, H. Ahmad, K. Khan, M. Hanif, H. Song, M. Alshamari, Y. Cao. "A Survey on Blockchain Technology: Evolution, Architecture and Security". In: *IEEE Access* PP (Apr. 2021), pp. 1–1. DOI: 10.1109/ACCESS.2021.3072849 (cit. on p. 20).

[BMM94] T. Berners-Lee, L. Masinter, M. McCahill. *Uniform Resource Locators (URL)*. 1994. URL: https://www.ietf.org/rfc/rfc1738.txt (cit. on p. 29).

[BMZ18] L. M. Bach, B. Mihaljevic, M. Zagar. "Comparative analysis of blockchain consensus algorithms". In: *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. 2018, pp. 1545–1550. DOI: 10.23919/MIPRO.2018.8400278 (cit. on p. 23).

[Buc16] E. Buchman. "Tendermint: Byzantine Fault Tolerance in the Age of Blockchains". In: 2016 (cit. on p. 21).

[CDE+16] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. Kosba, A. Miller, P. Saxena, E. Shi, E. Gün Sirer, D. Song, R. Wattenhofer. "On Scaling Decentralized Blockchains". In: *Financial Cryptography and Data Security*. Ed. by J. Clark, S. Meiklejohn, P. Y. Ryan, D. Wallach, M. Brenner, K. Rohloff. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 106–125. ISBN: 978-3-662-53357-4 (cit. on p. 23).

[Cer09] Certicom Research: Standards for Efficient Cryptography. *SEC 1: Elliptic Curve Cryptography*. 2009. URL: https://www.secg.org/sec1-v2.pdf (cit. on p. 54).

[CFI23] CFI Team. *Libra Cryptocurrency*. 2023. URL: https://corporatefinanceinstitute.com/resources/cryptocurrency/libra-cryptocurrency/ (cit. on p. 42).

[Cha21] Chainlink. *What Is a Blockchain Oracle?* 2021. URL: https://chain.link/education/blockchain-oracles (visited on 01/06/2023) (cit. on p. 34).

[CL99] M. Castro, B. Liskov. "Practical Byzantine Fault Tolerance". In: *OSDI* (Mar. 1999) (cit. on p. 21).

[Cos21] CosmWasm. *CosmWasm*. 2021. (Visited on 01/08/2023) (cit. on pp. 39, 40).

[Dan12] Q. Dang. *Secure Hash Standard (SHS)*. en. Mar. 2012. DOI: https://doi.org/10.6028/NIST.FIPS.180-4 (cit. on p. 24).

[DGL+16] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, L. Safina. *Microservices: yesterday, today, and tomorrow*. 2016. DOI: 10.48550/ARXIV.1606.04036. URL: https://arxiv.org/abs/1606.04036 (cit. on p. 76).

[Dur11] V. Durham. *Namecoin: A decentralized name registration system based on Bitcoin*. https://www.namecoin.org/. 2011 (cit. on p. 41).

[ETC16] ETC community. *Ethereum Classic*. https://ethereumclassic.org/. 2016 (cit. on p. 40).

[Eth21] Ethereum Foundation. *Ethereum Energy Consumption*. https://ethereum.org/en/energy-consumption/. [Accessed: February 18, 2023]. 2021 (cit. on p. 20).

[FBD+20] G. Falazi, U. Breitenbücher, F. Daniel, A. Lamparelli, F. Leymann, V. Yussupov. "Smart Contract Invocation Protocol (SCIP): A Protocol for the Uniform Integration of Heterogeneous Blockchain Smart Contracts". In: Lecture Notes in Computer Science 12127 (June 2020), pp. 134–149. DOI: 10.1007/978-3-030-49435-3_9 (cit. on pp. 15, 16, 26, 28, 51, 66).

[Gil18] M. Gill. *What is Proof of Burn?* [Online; accessed 21-February-2023]. 2018. URL: https://99bitcoins.com/what-is-proof-of-burn/ (cit. on p. 21).

[GY22] H. Guo, X. Yu. "A survey on blockchain technology and its security". In: *Blockchain: Research and Applications* 3.2 (2022), p. 100067. ISSN: 2096-7209. DOI: https://doi.org/10.1016/j.bcra.2022.100067. URL: https://www.sciencedirect.com/science/article/pii/S2096720922000070 (cit. on p. 20).

[Ham15] M. Hamburg. "EdDSA for more curves". In: *IACR Cryptology ePrint Archive* 2015 (2015), p. 618 (cit. on p. 45).

[HHS+20] A. Hentschel, Y. Hassanzadeh-Nazarabadi, R. Seraj, D. Shirley, L. Lafrance. *Flow: Separating Consensus and Compute – Block Formation and Execution*. 2020. DOI: 10.48550/ARXIV.2002.07403. URL: https://arxiv.org/abs/2002.07403 (cit. on p. 46).

[HHS20] A. Hafid, A. S. Hafid, M. Samih. "Scaling Blockchains: A Comprehensive Survey". In: *IEEE Access* 8 (2020), pp. 125244–125262. DOI: 10.1109/ACCESS.2020.3007251 (cit. on p. 23).

[HSL19] A. Hentschel, D. Shirley, L. Lafrance. *Flow: Separating Consensus and Compute*. 2019. DOI: 10.48550/ARXIV.1909.05821. URL: https://arxiv.org/abs/1909.05821 (cit. on pp. 40, 46).

[HSLZ19] A. Hentschel, D. Shirley, L. Lafrance, M. Zamski. *Flow: Separating Consensus and Compute – Execution Verification*. 2019. DOI: 10.48550/ARXIV.1909.05832. URL: https://arxiv.org/abs/1909.05832 (cit. on pp. 22, 46).

[Hyp22] Hyperledger Foundation. *Hyperledger Cactus Wiki*. Tech. rep. Hyperledger, 2022. URL: https://wiki.hyperledger.org/display/cactus (visited on 02/18/2023) (cit. on p. 33).

[Int15] Interchain GmbH. *Tendermint Core*. 2015. URL: https://github.com/tendermint/tendermint (visited on 01/08/2023) (cit. on pp. 21, 36, 49).

[JHGR20] S. Johannes, U. B. Hans, F. Gilbert, K. Robert. "The Energy Consumption of Blockchain Technology: Beyond Myth". In: *Business and Information Systems Engineering* 62 (2020), pp. 599–608. DOI: 10.1007/s12599-020-00656-x (cit. on pp. 15, 41).

[JL21] X.-J. Jiang, X. F. Liu. "CryptoKitties Transaction Network Analysis: The Rise and Fall of the First Blockchain Game Mania". In: *Frontiers in Physics* 9 (2021). ISSN: 2296-424X. DOI: 10.3389/fphy.2021.631665. URL: https://www.frontiersin.org/articles/10.3389/fphy.2021.631665 (cit. on p. 46).

[JSO10] JSON-RPC Working Group. *JSON-RPC 2.0 Specification*. 2010. URL: https://www.jsonrpc.org/specification (cit. on p. 66).

[KKZ19] K. Karantias, A. Kiayias, D. Zindros. *Proof-of-Burn*. 2019. URL: https://eprint.iacr.org/2019/1096.pdf (cit. on p. 21).

[KNAB17] J. Kanani, S. Nailwal, A. Arjun, M. Bjelic. *Polygon*. https://polygon.technology/. 2017 (cit. on p. 41).

[LFB+19] A. Lamparelli, G. Falazi, U. Breitenbücher, F. Daniel, F. Leymann. *Smart Contract Locator (SCL) and Smart Contract Description Language (SCDL)*. Oct. 2019 (cit. on pp. 29, 30).

[Lit11] Litecoin foundation. *Litecoin*. https://litecoin.org/. 2011 (cit. on p. 40).

[LSP02] L. Lamport, R. Shostak, M. Pease. "The Byzantine Generals Problem". In: *ACM Trans. Program. Lang. Syst.* 4 (Feb. 2002). DOI: 10.1145/357172.357176 (cit. on pp. 19, 22).

[LXS+19] Z. Liu, Y. Xiang, J. Shi, P. Gao, H. Wang, X. Xiao, B. Wen, Y.-C. Hu. "HyperService: Interoperability and Programmability Across Heterogeneous Blockchains". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (2019) (cit. on p. 35).

[MBH+22] H. Montgomery, H. Borne-Pons, J. Hamilton, M. Bowman, P. Somogyvari, S. Fujimoto, T. Takeuchi, T. Kuhrt, R. Belchior. *Hyperledger Cactus Whitepaper*. Mar. 2022. URL: https://github.com/hyperledger/cactus/blob/main/whitepaper/whitepaper.md (cit. on p. 33).

[MFH22] A. M. Mowad, H. Fawareh, M. A. Hassan. "Effect of Using Continuous Integration (CI) and Continuous Delivery (CD) Deployment in DevOps to reduce the Gap between Developer and Operation". In: *2022 International Arab Conference on Information Technology (ACIT)*. 2022, pp. 1–8. DOI: 10.1109/ACIT57182.2022.9994139 (cit. on p. 84).

[MMT+22] R. A. A. Mochram, C. T. Makawowor, K. M. Tanujaya, J. V. Moniaga, B. A. Jabar. "Systematic Literature Review: Blockchain Security in NFT Ownership". In: *2022 International Conference on Electrical and Information Technology (IEIT)*. 2022, pp. 302–306. DOI: 10.1109/IEIT56384.2022.9967897 (cit. on p. 25).

[Mou16] W. Mougayar. *The Business Blockchain: Promise, Practice, and Application of the Next Internet Technology*. 1. John Wiley and Sons, May 2016 (cit. on pp. 15, 17, 19).

[Mys22] MystenLabs. *Sui*. 2022. URL: https://sui.io/ (visited on 02/25/2023) (cit. on p. 40).

[Nak08] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. https://bitcoin.org/bitcoin.pdf. 2008 (cit. on pp. 15, 20).

[NEA18] NEAR Foundation. *NEAR Protocol*. https://near.org/. 2018 (cit. on p. 41).

[Noy16] C. Noyes. *BitAV: Fast Anti-Malware by Distributed Blockchain Consensus and Feedforward Scanning*. 2016. DOI: 10.48550/ARXIV.1601.01405. URL: https://arxiv.org/abs/1601.01405 (cit. on p. 15).

[Opt20] Optimism Foundation. *Optimism*. https://optimism.io/. 2020 (cit. on p. 41).

[PDS14] R. Plösch, A. Dautovic, M. Saft. "The Value of Software Documentation Quality". In: *2014 14th International Conference on Quality Software*. 2014, pp. 333–342. DOI: 10.1109/QSIC.2014.22 (cit. on p. 38).

[Pee12] Peercoin foundation. *peercoin*. https://www.peercoin.net/. 2012 (cit. on p. 40).

[PKF+] S. Park, A. Kwon, G. Fuchsbauer, P. Gaži, J. Alwen, K. Pietrzak. *SpaceMint: A Cryptocurrency Based on Proofs of Space*. URL: https://eprint.iacr.org/2015/528.pdf (cit. on p. 21).

[PPC15] G. Peters, E. Panayi, A. Chapelle. "Trends in Crypto-Currencies and Blockchain Technologies: A Monetary Theory and Regulation Perspective". In: *SSRN Electronic Journal* (Sept. 2015). DOI: 10.2139/ssrn.2646618 (cit. on p. 15).

[Pri13] Primecoin community. *Primecoin*. https://primecoin.io/. 2013 (cit. on p. 40).

[R316] R3. *Corda*. https://corda.net/. 2016 (cit. on p. 41).

[RJCe22] Randall-Mysten, Jibz1, Clay-Mysten, econmysten. *Proof of Stake*. Last update: 1/25/2023, 1:20:13 AM. 2022. URL: https://docs.sui.io/learn/tokenomics/proof-of-stake#sui-token-delegation (visited on 02/18/2022) (cit. on p. 20).

[RSA78] R. L. Rivest, A. Shamir, L. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". In: *Commun. ACM* 21.2 (Feb. 1978), pp. 120–126. ISSN: 0001-0782. DOI: 10.1145/359340.359342. URL: https://doi.org/10.1145/359340.359342 (cit. on p. 17).

[Sav17] A. Savelyev. "Contract law 2.0: 'Smart' contracts as the beginning of the end of classic contract law". In: *Information & Communications Technology Law* 26.2 (2017), pp. 116–134. DOI: 10.1080/13600834.2017.1301036. eprint: https://doi.org/10.1080/13600834.2017.1301036. URL: https://doi.org/10.1080/13600834.2017.1301036 (cit. on p. 23).

[SHA+] C. Smith, HaoTian, E. Awosika, R. Pujari, ethosdev, P. Wackerow, Y. Yadav, Joshua, J. Cook, S. A. Green, M. Havel, J. Degesys, S. Richards, selfwithin anbd Victor Luna, A. Ismodes, R. Cordell, tentodev, Alwin. *PROOF-OF-STAKE (POS)*. URL: https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/ (cit. on pp. 20, 42).

[SK22] B. Sriman, S. G. Kumar. "Decentralized finance (DeFi): The Future of Finance and Defi Application for Ethereum blockchain based Finance Market". In: *2022 International Conference on Advances in Computing, Communication and Applied Informatics (ACCAI)*. 2022, pp. 1–9. DOI: 10.1109/ACCAI53970.2022.9752657 (cit. on p. 42).

[Slo23] SlowMist. *Blockchain Security and AML Analysis report*. 2023. URL: https://www.slowmist.com/report/2022-Blockchain-Security-and-AML-Analysis-Annual-Report(EN).pdf (cit. on p. 42).

[SN17] A. J. Steve Ellis, S. Nazarov. *ChainLink A Decentralized Oracle Network*. 2017. URL: https://research.chain.link/whitepaper-v1.pdf (visited on 01/06/2023) (cit. on p. 34).

[Sol20] Solana Foundation. *Solana*. https://solana.com/. 2020 (cit. on p. 41).

[Sta21] Starknet community. *Starknet*. https://www.starknet.io/. 2021 (cit. on p. 40).

[Ste14] Stellar Development Foundation. *Stellar*. https://www.stellar.org. 2014 (cit. on p. 22).

[Sui17] D. Suiu. *Plugin Framework for Java*. 2017. URL: https://pf4j.org/ (visited on 01/21/2023) (cit. on p. 73).

[Sza96] N. Szabo. "Smart Contracts: Building Blocks for Digital Markets". In: (1996). URL: https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html (cit. on p. 23).

[Tea14] M. D. Team. *Mazacoin*. https://www.mazacoin.org/. 2014 (cit. on p. 41).

[Ten19] Tendermint. *Cosmos SDK*. https://v1.cosmos.network/sdk. 2019 (cit. on p. 40).

[The17] The ZILLIQA Team. *The ZILLIQA Technical Whitepaper*. 2017. URL: https://docs.zilliqa.com/whitepaper.pdf (cit. on p. 21).

[Tro17] Tron Foundation. *Tron*. https://tron.network/. 2017 (cit. on p. 41).

[TSH22] L. T. Thibault, T. Sarry, A. S. Hafid. "Blockchain Scaling Using Rollups: A Comprehensive Survey". In: *IEEE Access* 10 (2022), pp. 93039–93054. DOI: 10.1109/ACCESS.2022.3200051 (cit. on pp. 23, 38).

[TT19] P. Tasca, C. J. Tessone. "A Taxonomy of Blockchain Technologies: Principles of Identification and Classification". In: *Ledger* 4 (Feb. 2019). DOI: 10.5195/ledger.2019.140. URL: https://ledger.pitt.edu/ojs/ledger/article/view/140 (cit. on pp. 17, 20, 21, 24).

[Web17] Web3 foundation. *Polkadot*. https://polkadot.network/. 2017 (cit. on p. 40).

[Wik22] Wikipedia contributors. *Byzantine fault*. Dec. 2022. URL: https://en.wikipedia.org/wiki/Byzantine_fault (cit. on p. 45).

[Wik23] Wikipedia. *List of blockchains*. 2023. URL: https://en.wikipedia.org/wiki/List_of_blockchains (visited on 02/18/2023) (cit. on p. 37).

[Woo14] G. Wood. *Less-techy: What is Web 3.0?* Apr. 2014. URL: http://gavwood.com/web3lt.html (visited on 01/17/2023) (cit. on p. 24).

[WSW20] Y. Wu, P. Song, F. Wang. "Hybrid Consensus Algorithm Optimization: A Mathematical Method Based on POS and PBFT and Its Application in Blockchain". In: *Mathematical Problems in Engineering* 2020 (Apr. 2020), pp. 1–13. DOI: 10.1155/2020/7270624 (cit. on p. 21).

[ZW15] Y. Zhang, J. Wen. "An IoT electric business model based on the protocol of bitcoin". In: *2015 18th International Conference on Intelligence in Next Generation Networks*. 2015, pp. 184–191. DOI: 10.1109/ICIN.2015.7073830 (cit. on p. 15).

[ZXD+18] Z. Zheng, S. Xie, H.-N. Dai, X. Chen, H. Wang. "Blockchain challenges and opportunities: A survey". In: *International Journal of Web and Grid Services* 14 (Oct. 2018), p. 352. DOI: 10.1504/IJWGS.2018.095647 (cit. on p. 15).

All links were last followed on February 22, 2023.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature