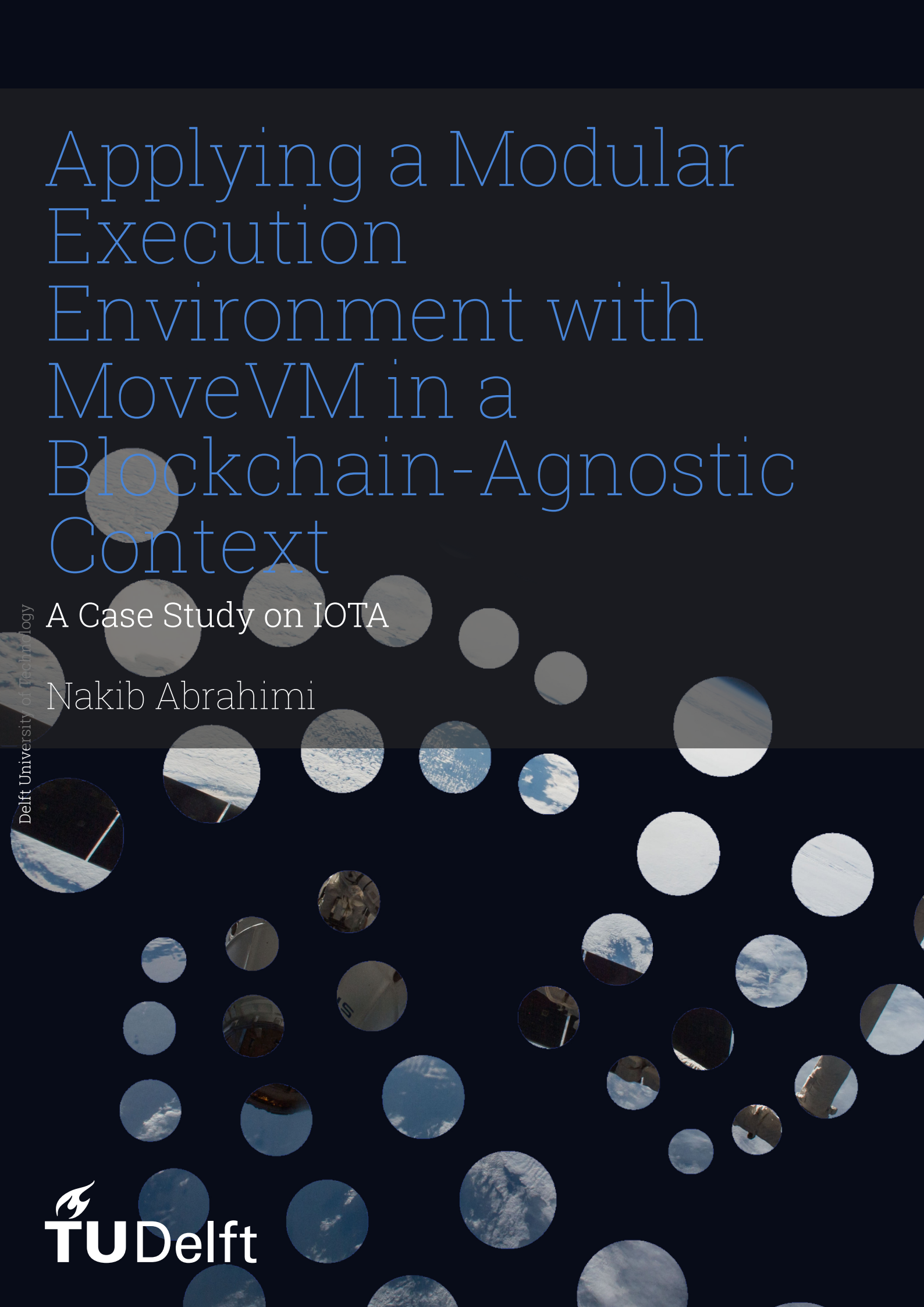
Applying a Modular Execution Environment with MoveVM in a Blockchain-Agnostic Context

A Case Study on IOTA

Nakib Abrahimi

Delft University of Technology



TU Delft

Applying a Modular Execution Environment with MoveVM in a Blockchain-Agnostic Context

A Case Study on IOTA

by

Nakib Abrahimi

Nakib Abrahimi 4488849

Thesis Advisor:	Jérémie Decouchant
Company Supervisor:	Levente Pap
Committee Member:	Kaitai Liang
Project Duration:	November, 2023 - July, 2024
Faculty:	EEMCS, Distributed Systems, Delft

TU Delft

Preface

This thesis represents the work of my research journey into MoveVM with the IOTA team. I would like to express my deepest gratitude to my thesis advisor, Jérémie Decouchant, and company supervisor, Levente Pap. Their guidance and support were invaluable throughout the thesis period. I could not have wished for anything more.

Special thanks to Can Umut Ileri for introducing me to the team at IOTA and informing me about their research on Move, which was coincidentally my interest at the time.

I am also thankful to my dear colleagues at IOTA, who provided me with invaluable feedback and support, especially Mirko Zichichi, whose expert insights and feedback were particularly helpful.

Lastly, I would also like to thank my family and friends for their support and encouragement. Without the people around me, finishing this thesis would have been a much more difficult task.

Nakib Abrahimi
Delft, July 2024

Abstract

This thesis explores the application of a modular execution environment, specifically utilizing the Move Virtual Machine (MoveVM), within a blockchain-agnostic framework. The study aims to demonstrate how this modular approach can enhance the execution capability of existing blockchain systems. The case study focuses on the IOTA Distributed Ledger Technology (DLT), known for its unique Tangle architecture, which differentiates it from traditional blockchain technologies.

The research begins by detailing the current limitations in existing blockchain platforms, such as their dependence on specific consensus algorithms and rigid execution environments. It then introduces the MoveVM, developed firstly by the Libra (Diem) project and now by the projects Sui and Aptos, highlighting its advantages in terms of security, programmability, and modularity.

By integrating an object-flavored MoveVM into the IOTA framework, the study examines how the modular smart contract execution environment can operate almost independently of the underlying ledger. The work for this thesis was conducted in two main parts. In the first part, a prototype was created by integrating a modified version of the existing IOTA node software with an object-flavored Move execution environment. In the second part, a Move Swap smart contract was developed at the application layer to showcase the system's ability to support an advanced intent-based architecture. This approach, which executes based on user intents rather than declarative smart contract commands, offers significant economic benefits and reduces unnecessary costs for users.

To validate the effectiveness of this approach, the thesis presents empirical data and performance metrics gathered from various test scenarios. The results demonstrate the successful integration of the Move smart contract execution into the IOTA node software, but also significant improvements in resource efficiency thanks to the intent-based architecture.

In conclusion, this thesis contributes to the growing body of knowledge on blockchain technology by showcasing the potential of MoveVM in enhancing the functionality and performance of blockchain-agnostic networks. Moreover, the findings suggest that the intent-based approach not only simplifies user interactions but also enables advanced functionalities and custom on-chain VMs, paving the way for innovative applications in DLTs.

Contents

List of Figures

List of Tables

1

Introduction and Background

1.1. Definition and Characteristics of Blockchains

1.1.1. Blockchain technology

Blockchain technology first made its entrance into the world in 2008, with the release of the groundbreaking whitepaper by Satoshi Nakamoto: ”Bitcoin: A Peer-to-Peer Electronic Cash System” [1]. This whitepaper introduced the concept of a decentralized digital currency that allows for secure, transparent and immutable transactions between parties without any intermediate parties such as banks. It manages to do so by making use of blockchain technology, which is a distributed set of computers that maintain a shared ledger (state). The ledger is updated by transactions, which first have to go through a consensus mechanism, ensuring that all participants have a synchronized, verified, and accurate record of transactions.

A blockchain is a type of Distributed Ledger Technology (DLT). It should be noted that a blockchain is not the only type of DLT available. A blockchain is a type of DLT with a linear data structure in which blocks of packed transaction data are chained to each other. All blocks, except the first genesis block, contain the hashes of the block that comes before it, hence the concept of a chain of blocks. Other types of DLT include, but are not limited to, Directed Acyclic Graphs (DAGs) [2] and Hashgraphs [3]. The data architectures of these three types of DLTs are visualized in fig 1.1.

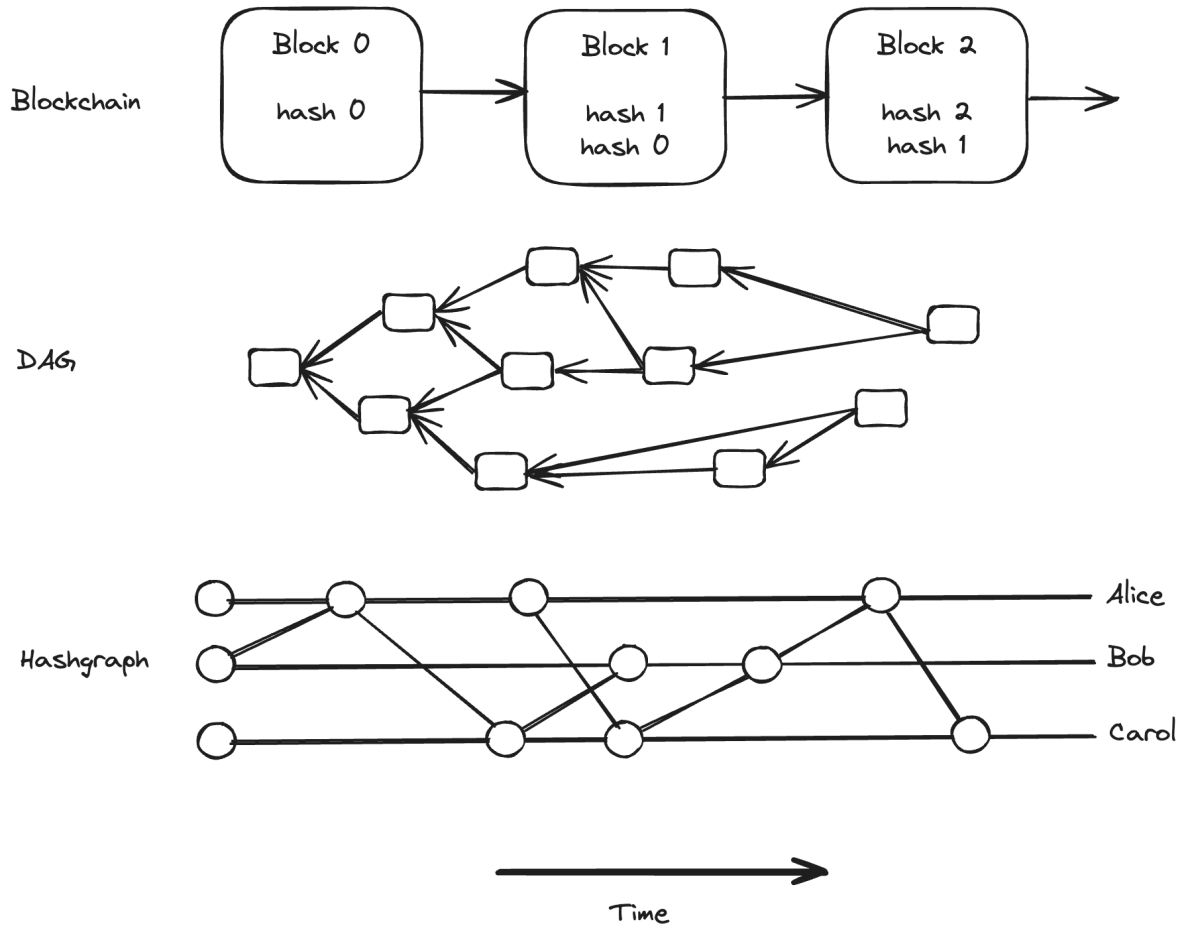


Figure 1.1: Different types of DLTs

1.1.2. Advantages of Distributed Systems

Distributed systems like blockchains have many benefits in regards to reliability, transparency, and security. One such benefit is the inherent ability to be resilient to single points of failure. Blockchains, and DLTs in general, are designed to operate over a network of distributed nodes. Each of these nodes contains a full copy of the network state, which makes the state redundant. In most blockchains, all transactions are transparent and immutable, which are available to all the network participants, even those that do not run a local full node which holds the full transaction history [4]. This transparency decreases the level of fraud and manipulation in the system, as any fraudulent or non-wanted activity can be spotted.

It is possible that such a distributed blockchain system could have prevented The Great Recession in 2008. It is general consensus that the cause of this financial crisis was the freezing of the international inter-bank market in August 2007, which was the result of mutual distrust within the banking industry. This turned out to be valid as banks were indeed insolvent [5].

1.2. Programmability in Blockchains

A blockchain without user programmability is sufficient in case that is the only purpose of the chain. However, with what we have seen in the early days of the Internet, where the first widespread use cases of it were static websites [6]. With the arrival of JavaScript and Flash, much more things were possible on the web, such as dynamic and interactive user interfaces [7, 8]. In today's society, the programmability introduced by JavaScript to the static web cannot be overlooked. [9].

A DLT is powerful as it is, and especially Bitcoin and blockchain technology. However, the addition of programmability in blockchains opens up a whole new world of possibilities. Boring distributed ledgers turn into bustling dynamic platforms for decentralized applications (dApps) and smart contracts. The idea behind programmability in blockchains is the ability to execute code on the blockchain. This enables the creation of smart contracts, which automate terms and conditions between parties, without any intermediary. Nick Szabo, the inventor of the term, would illustrate it by comparing it to a vending machine [10]. Inserting the right amount of coins will make the machine deliver the requested goods, without trusting an intermediary, and the technological infrastructure of the machine is a guarantee that this contract will be honored exactly as planned. With programmability in blockchains, a blockchain will extend beyond the initial capability of only recording transactions.

As of date, smart contracts have introduced numerous innovative elements to today's society. They have brought us tokenization of real world assets, programmable money, streamlined automated processes, Decentralized Autonomous Organizations (DAOs) and most notably, Decentralized Finance (DeFi) [11]. Decentralized Exchanges (Dexs) have a high volume throughput, peaked at 235 billion USD in the month November of 2021, as seen in figure 1.2. The ability to facilitate such volume with little to no maintenance on its code and small project teams is what makes this feat truly revolutionary.

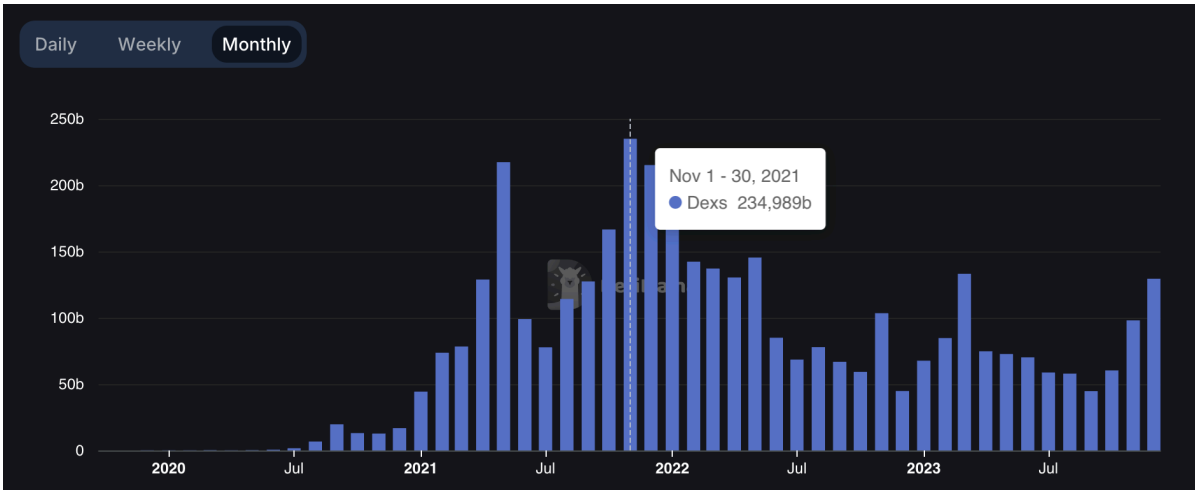


Figure 1.2: Monthly volume of Dexs. Source: DefiLlama.com

1.3. Blockchain Architecture

A typical blockchain consists of multiple interconnected machines which are called nodes, as seen in figure 1.3, with each node consisting of a number of core components [12].

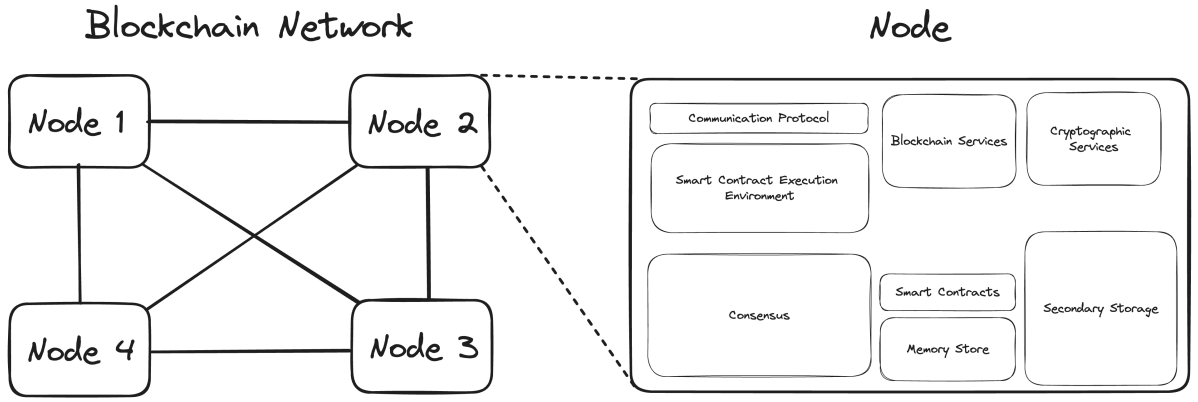


Figure 1.3: Example blockchain network with 4 nodes.

1. **Secure Runtime Environment**
   Certain transactions might have to deal with smart contract functions that require to be executed in a secure environment. This is generally called the Smart Contract Execution Environment, or Virtual Machine (VM).
2. **Cryptographic Services**
   Cryptographic algorithms like hashing functions and digital signatures.
3. **Smart Contracts**
   Smart contracts that are deployed on the chain and can be interacted with. Uses the Secure Runtime Environment to execute functions on the node.
4. **Blockchain Secondary Storage**
   To store all ledger information, such as transactions, but also the smart contracts.
5. **Blockchain memory store**
   Stores the latest transactions in memory for fast retrieval and execution of transactions. This includes the transaction mempool, which is a collection of submitted transaction, but not yet executed or agreed upon.
6. **Consensus protocol**
   Consensus protocol like Proof of Work (PoW), Proof of Stake (PoS) and practical Byzantine Fault Tolerance (pBFT). Decides how the nodes agree on the validity of transactions and which transactions enter the blockchain state and in which order.
7. **Blockchain Services**
   Blockchain specific additional services, such as membership services in a permissioned blockchain. Could contain APIs as well.
8. **Communication protocol**
   The communication protocol that nodes use to communicate with each other, for example HTTP(S) and gRPC.

1.4. Introduction to Ethereum

1.4.1. Ethereum Overview

The first blockchain platform that facilitated programmable money through smart contracts was Ethereum, launched in 2015 and created by Vitalik Buterin [13]. Ethereum smart contracts are written primarily in a high level programming language called Solidity, which has syntactical resemblance with JavaScript. This high level language is then compiled into EVM bytecode, which is run on the Ethereum Virtual Machine (EVM).

A part of this bytecode, the runtime bytecode, is deployed on the Ethereum chain and known as the smart contract. Other (user)parties are then able to interact with the functions of this smart contract through transactions, or simply view the data stored in the smart contract.

As of date, Ethereum has a vast ecosystem of dApps and has the most users and developers, far exceeding all other smart contract platforms. The reason for this position could be due to early mover advantage.

Transactions cause world state modifications, as illustrated in 1.4. When a transaction is included in the next block and executed, it triggers changes to the state of the system. These changes could be value transfers between accounts, execution of smart contract functions or creation of new contracts altogether. Each of these transactions are recorded on the blockchain and contribute to the total history of state transitions.

1.4.2. Turing Completeness and Risks

Ethereum's Solidity is Turing complete, meaning that the smart contracts are capable of performing any task a computer could normally perform. It allows developers to be flexible, so that any system can be built using Ethereum, such as custom coins, on-chain physical assets, non-fungible assets (NFTs) and numerous variants of dApps.

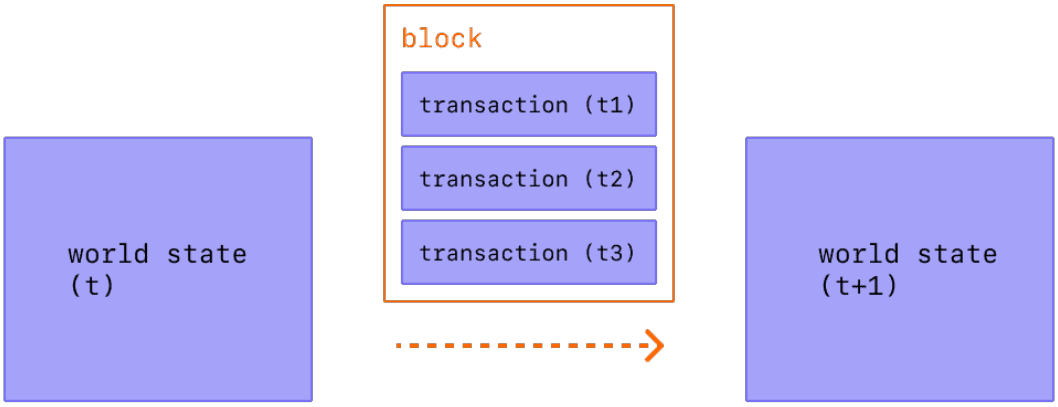


Figure 1.4: World state transition. Source: medium.com/cybermiles

However, being Turing-complete also comes with some risks, due to its possible complexity. Not only does this make it harder for smart contract developers to write safe code, but also makes it harder to audit the code. This design aspect has led to security issues on Ethereum and other EVM-based chains, ultimately causing massive financial losses as seen in figure 1.5 and figure 1.6.

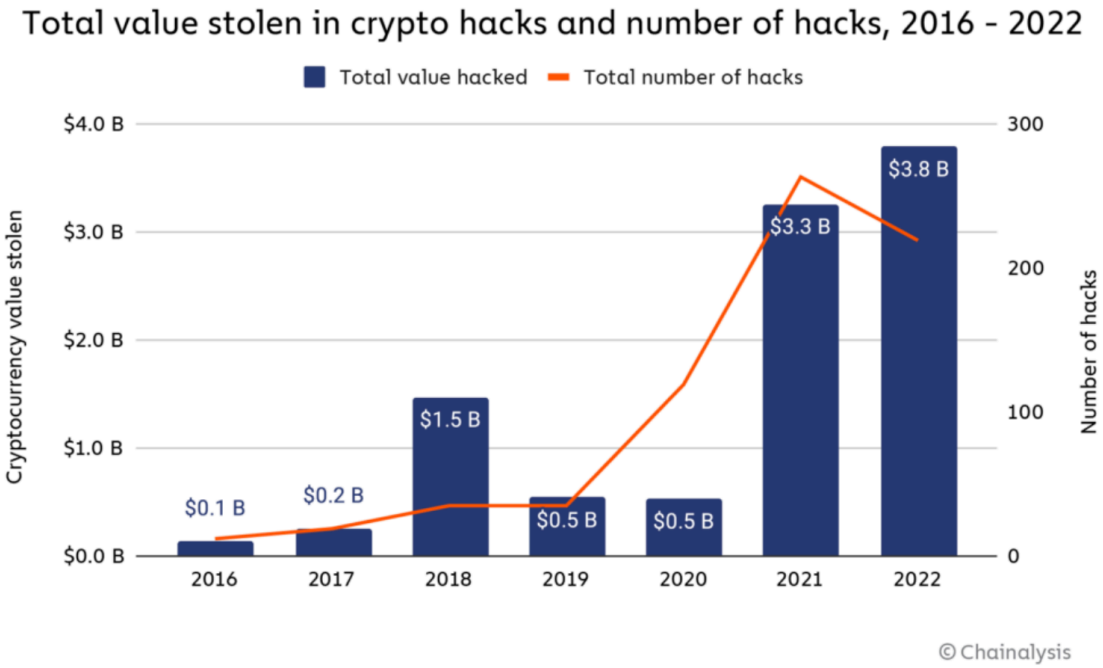


Figure 1.5: Value stolen, Source: Chainanalysis

A notable amount of these stolen funds come from Decentralized Finance (DeFi) apps. These apps

are essentially smart contracts, so the value is stolen from the smart contracts.

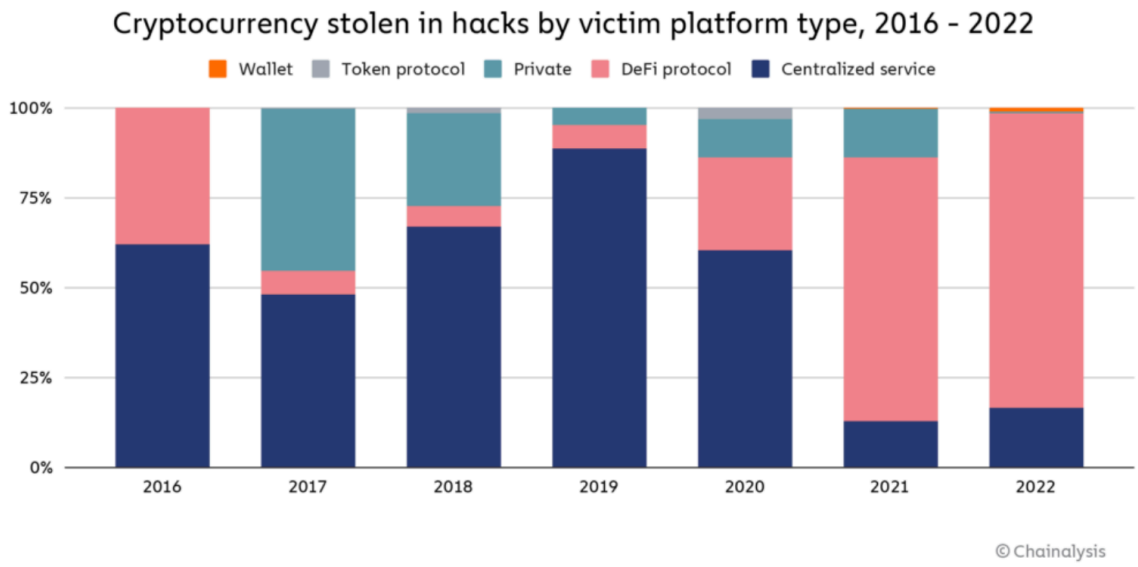


Figure 1.6: Value stolen categories, Source: Chainanalysis

1.5. Custom VMs

After Ethereum, many other new projects adopted the EVM to achieve blockchain programmability, i.e. Binance Smart Chain and Fantom [14]. This has proven to be a successful strategy for many of the projects, as the EVM has all the necessary documentation, tools, developers and general infrastructure with it. Also, thanks to (token) standards [15], it became easier to transfer (bridge) tokens from Ethereum to other EVM based chains that enforce the same standards. Not only that, but due to an identical virtual machine (VM), all smart contracts written for one EVM chain are fully compatible with the other. This essentially makes onboarding of developers and users more convenient for new chains, and should make the overall user experience better.

Despite this network benefit of EVM, many other projects decided to develop their own VM due to the inability for EVM to mold into their completely new chain [16]. It was necessary for them to create a custom VM, which does not carry the burden of the EVM design decisions, as the EVM was specifically built for Ethereum. This is usually a last resort, as writing your own programming language or Domain-specific Language (DSL) and VM is time and resource consuming, especially when there is already a working product that could be used.

One of the reasons a new blockchain could opt for a new custom VM and its associated programming language is the previously mentioned burden aspect. A possible burden could be EVMs reliance on one type of native GAS token to pay for EVM code execution. Each transaction on the EVM comes with a required gas fee, depending on the amount of calculations the EVM has to make, that is paid by the transaction issuer. A blockchain adopting EVM is therefore obliged to pay for code execution by one native token.

Other reasons for custom VMs could be to create a more optimized language and VM, tailored to the rest of the components that form the blockchain architecture such as the consensus layer.

1.6. Thesis Overview

This thesis explores the integration of the Move Language and Move Virtual Machine (MoveVM) within the IOTA blockchain to facilitate Layer 1 smart contracts. The integration is structured into two primary phases, and concludes with a prototype of an intent-based decentralized app built on the integration.

In the initial development phase, the focus is on developing an adapter to enable the MoveVM to operate within the IOTA environment. This involves creating a confined setup using a mock node to simulate the IOTA network. Key tasks include implementing the adapter for MoveVM, ensuring compatibility and smooth operation within the mock node setup, and conducting extensive testing to validate functionality in this controlled environment.

The second development phase transitions from the mock node to the actual IOTA network by integrating with a Hornet Node, which includes the consensus mechanism. This phase involves replacing the mock node with the Hornet Node, testing and validating the MoveVM functionality within the real IOTA network, and ensuring seamless operation with IOTA's consensus and transaction mechanisms.

Following successful integration, the thesis proceeds to develop a dApp prototype demonstrating intent-based execution on the IOTA network. This includes designing and implementing a simple prototype, showcasing the efficiency of intent-based execution and a proper functioning of the new execution platform.

The final section also presents a more advanced design for an intent-based execution pipeline, proposing enhancements and optimizations based on the prototype's performance, discussing potential scalability solutions and future developments.

This research aims to bridge the capabilities of the Move Language and MoveVM with the IOTA blockchain, providing a robust platform for Layer 1 smart contracts. Through the two-phase development process and the creation of a prototype dApp, this thesis demonstrates the feasibility and advantages of this integration, paving the way for future advancements in blockchain technology.

1.7. Related Work

In the course of this research, several other execution platforms were considered as potential candidates for IOTA. Each of these VMs are built for ledgers that utilize a UTXO-based accounting system, and are therefore worth mentioning in the context of developing Layer 1 smart contracts on the IOTA UTXO-based DLT.

1.7.1. Cardano's Plutus

Cardano makes use of an extended UTXO based ledger, which aims at higher expressiveness of programmability while maintaining all the benefits of Bitcoin's UTXO model [17].

Plutus (Core) is a native smart contract language for Cardano. It is a Turing-complete language based on Haskell, a pure functional programming language, so Plutus smart contracts are effectively Haskell programs [18].

1.7.2. Bitcoin Script

Bitcoin users interact with the system via addresses. Transfers of Bitcoins between these addresses are depicted as transactions. Typically, a transaction references previous transaction outputs, known as Unspent Transaction Outputs (UTXOs), as new transaction inputs and directs all input Bitcoin values to new outputs. The logic for linking inputs to UTXOs is defined by programmable functions called scripts. Bitcoin Script is essentially a collection of instructions stored with each new transaction. These instructions specify how users can access and utilize the bitcoins available on the network. A Bitcoin transaction comprises a set of data for n input transactions and m output transactions. For each output transaction, a lock script specifies what actions are required to use that output in the future. For each input transaction, an unlock script specifies what is done to spend the referenced output.

1.7.3. FuelVM

The Fuel VM is a custom VM built for Fuel, and uses Sway for its smart contracts. It is designed to be modular, so it can be used as the execution environment for any blockchain [19].

The VM is optimizes for higher throughput of transaction execution. It is UTXO based and requires each transaction to define the UTXOs that it will touch. In this way, it can parallelize the transactions that do not interfere with each other.

1.7.4. Radix Engine

The Radix Engine is a custom execution environment built for the Radix DLT and uses the custom programming language Scrypto for its smart contracts. [20]

The core difference between Radix and other VMs is that Radix makes use of well-structured Final State Machines (FSMs) to handle tokens and assets, called resources. Radix has adopted a strategy where resources are an integral part of the platform, instead of being repeatedly implemented at the smart contract layer.

2

Background on Move

2.1. Move Language and MoveVM

2.1.1. Introduction

In this fast-evolving landscape, it becomes clear that it is important in which language a smart contract is written. This shapes the security, efficiency and adaptability of the numerous dApps that are written not only by senior programmers, but also novice ones. To grow the ecosystem, more developers need to be onboarded, and those include junior programmers as well. As the saying goes, a chain is only as strong as its weakest link. A language that connects with both levels of seniority is essential to strengthen the overall blockchain security and growth.

Move has entered the space as a domain specific language (DSL), which is specifically designed for programming assets on blockchains. Its design philosophy prioritizes first-class assets, flexibility, safety, and verifiability, which makes it great to develop reliable and robust dApps [21].

Sam Blackshear is widely known to be the creator of Move [22]. It was a core part of Libra, Facebooks new blockchain based payment network initiated in 2018. The goal was to create a safe and flexible programming language for smart contracts on Libra. It was designed with a focus on scarcity, which allows for reliable and secure smart contracts. It has features like resource types, which help prevent bugs and vulnerabilities are typically found in other smart contract languages.

- **First-class assets:** One of Move's main features is the ability to create custom resource types, embodying semantics inspired by linear logic [23]. These resource types have a fundamental property: they cannot be copied or implicitly discarded, but are exclusively movable between program storage locations. This concept has a high resemblance with Rust's ownership and borrowing system. In Rust, ownership ensures that data is managed safely and prevents dangling pointers or memory leaks [24].

  In DLTs utilizing Move, the main coin which is used for gas payments is implemented as a regular Move resource, with no special unique features in the language. These resources can be created, modified and destroyed in so called modules. Move modules are equivalent to smart contracts in other blockchain languages. Resources are thus integrated at the type level, instead of simply only supporting one resource value (e.g. Ether). This helps Move to stay blockchain-agnostic. Developers can quickly enjoy these benefits in custom assets without having to go through additional reimplementation processes needed for ERC20 [25] and such.

- **Flexibility:** Move modules provide a level of flexibility by allowing secure yet flexible code composition. At a broad level, the association between modules, resources and procedures in Move mirrors that of classes, objects, and methods in object-oriented programming.

  Move adds to this flexibility its built-in transaction scripts. These transaction scripts make it possible to call multiple procedures of modules in a single transaction. In other blockchains like Ethereum, this would require a separate smart contract. In Move, this is built-in and can be in-

voked with a single transaction, reducing gas fees as compared to when these transactions would be submitted individually.

- **Safety:** Move's executable format consists of typed bytecode, which has a higher level of abstraction from the machine code than assembly, but is lower than that of normal source code such as Rust, Java, etc. This bytecode is then undergoing checks on-chain for resource, type, and memory safety by the so called bytecode verifier, after which it is directly executed by the bytecode interpreter.
- **Verifiability:** Move performs light on-chain verification of key safety properties, and supports advanced off-chain static verification tools. Move has been designed with the following key design decisions in mind to make Move more approachable to static verification than most general programming languages:
  - *Static dispatch:* The target of each call location can be statically determined. Most other languages are using dynamic dispatch, which makes it hard to determine the call locations. This allows for verification tools to know what a call is actually doing, without performing complex and expensive call reconstruction analysis.
  - *Limited mutability:* Move's bytecode verifier uses a process similar to Rust's borrow checker to guarantee that there will always be at most one mutable reference to a used value.
  - *Modularity:* Move modules ensure that data is kept private (abstracted) and that all resource actions such as modifications are managed within the module itself. Because of this encapsulation, as well as Move's type protection, the rules set for a module's data cannot be violated by code from the outside. This all enables complete verification of a module's rules by evaluating it in isolation without considering how it interacts with outside programs.

This language was also designed with blockchain agnosticism in mind, which makes it possible for other chains to adopt this as a programming language for their smart contracts.

2.1.2. Move Lang Features

Resource Types

One of Move's main features is the ability to create custom resource types, embodying semantics inspired by linear logic. These resource types have a fundamental property: they cannot be copied or implicitly discarded, but are exclusively movable between program storage locations. This concept has a high resemblance with Rust's ownership and borrowing system. In Rust, ownership ensures that data is managed safely and prevents dangling pointers or memory leaks. Another way of seeing this concept is like a value conservation guarantee similar to (e.g.) the conservation of mass.

In DLTs utilizing Move, the main coin which is used for gas payments is implemented as a regular Move resource, with no special unique features in the language. These resources can be created, modified and destroyed in so called modules. Move modules are equivalent to smart contracts in other blockchain languages.

Move Type System

What gives the resources its powerful use, is the Move type system. It prevents misuse of resources, just like in the real world: you cannot simply duplicate, reuse, or discard physical assets. The type system ensures that digital assets behave like physical ones.

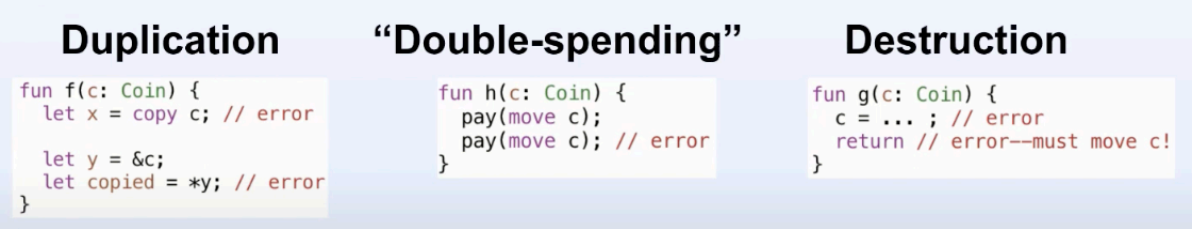


Figure 2.1: Type system protects against these cases, Source: Mysten Labs

Duplication of a resource (Coin in figure 2.1) is not allowed and returns an error at the type system level. Sneaky copying by copying the value of a reference is also not allowed. You also cannot use the same resource twice, as the resource has been consumed by the first pay function in fun h. And finally, destruction, as in fun g, is also not allowed. In this function, you simply overwrite the resource with another value and return before consuming this resource, which are both type system errors.

Such a resource type can only be created or deleted by the module that defines it. All these type guarantees are enforced statically by the Move virtual machine via bytecode verification. The Move virtual machine will refuse to run code that has not passed through the bytecode verifier.

Static typing

Worth mentioning is the static type system of Move. All variable types are known at compilation, which prevents data type errors during run-time. Examples of other statically typed programming languages are C, C++, Java and Solidity. These are safer than their counterpart: the dynamic type system. In these systems, the types are not known during compilation, only during run-time. Examples of dynamically typed programming languages are JavaScript and Python. In general, smart contract languages are statically typed, as dynamic typing would cause unexpected run-time issues, which in blockchains is not something you want to have.

Static dispatch

There is no dynamic dispatch, which means that for every function call the target is statically known. This implies that there is no re-entrancy risk [26] possibility since that requires dynamic dispatch, which is the case for many smart contract languages currently in existence such as Solidity for EVM. If you send a transaction which calls a function, you will know exactly which function in the code is called before the transaction is run. There is no possibility to inject code in between.

Bytecode Verification

The Move bytecode verifier is a verification system that has the following checks:

1. Type safety
2. Ability safety
3. Reference safety
   The equivalnt to the rust borrow checker, but at the bytecode level. No dangling references, no memory leaks and referential transparency.
4. Checks on control flow
   To make sure that the control-flow graph is reducable.
5. Locals safety
   Checking for nulled references, making sure that if you access a local that it is not moved yet (MoveLoc) or that it might be empty.
6. Stack balancing analysis
   Callee cannot touch caller's stack. The operating stack is shared across multiple procedures, so it is important to ensure that the callee cannot modify the values on the caller's stack. The caller might have money that they do not want to give to the callee for example. So this stack balancing check makes sure that each function call has its own portion of the stack, that does not belong to other functions.

This bytecode verifier is strong in a sense that it protects the programmer from themselves and from other programmers.

Code reusability

Another feature of Move is its code reusability. Move makes use of modules, and it is possible to reuse types and functions from other modules via imports. Move also has Generics, similar to the ones in Rust, which allows for code to be reused with different type parameters.

Abilities

Abilities are a feature in Move that allows you to have more control over what actions are permitted for values of a specific kind. In early versions of Move, there were only copyable values and resource values. The latter being types that could not be copied and had to be used. After the realization that more fine grained control was needed for some cases, a new type control system was needed: the abilities system [27].

These abilties can be annotated on structs in the code to add more fine grained access control:

- **copy**: Allows values of types with this ability to be copied.
- **drop**: Allows values of types with this ability to be popped/dropped.
- **store**: Allows values of types with this ability to exist inside a struct in global storage.
- **key**: Allows the type to serve as a key for global storage operations.

Resources, for example, only have the key and store ability. They cannot be copied or dropped.

Formal verification

Formal verification is the process of checking whether a design satisfies some requirements (properties).

The Move prover is a formal verification tool for smart contracts (modules) [28]. It allows developers to mathematically prove certain specified properties of the functions in their modules. These properties are usually specified by the module developer themself. The move prover runs for each of the specified functions before deployment of the modules. This is different from runtime assertions or the bytecode verifier. These last two have an impact during the runtime itself.

The Move prover uses a classical (Floyd/Hoarde) approach with explicit specifications.

- They are mathematically precise, written in their own specification language which is a logical language.
- They are separate from implementations.
- They explicitly capture user intent.

Formal verification automatically proves that Move programs satisfy specifications for all inputs, in all states. The goal is to prove correctness, it is not a bug hunting process. Errors in these specifications are dangerous. They may result in false positives, or false negatives which are missed errors, the more serious kind. So having a good and clear way to form these specifications is important.

The move prover architecture can be found in figure 2.2.

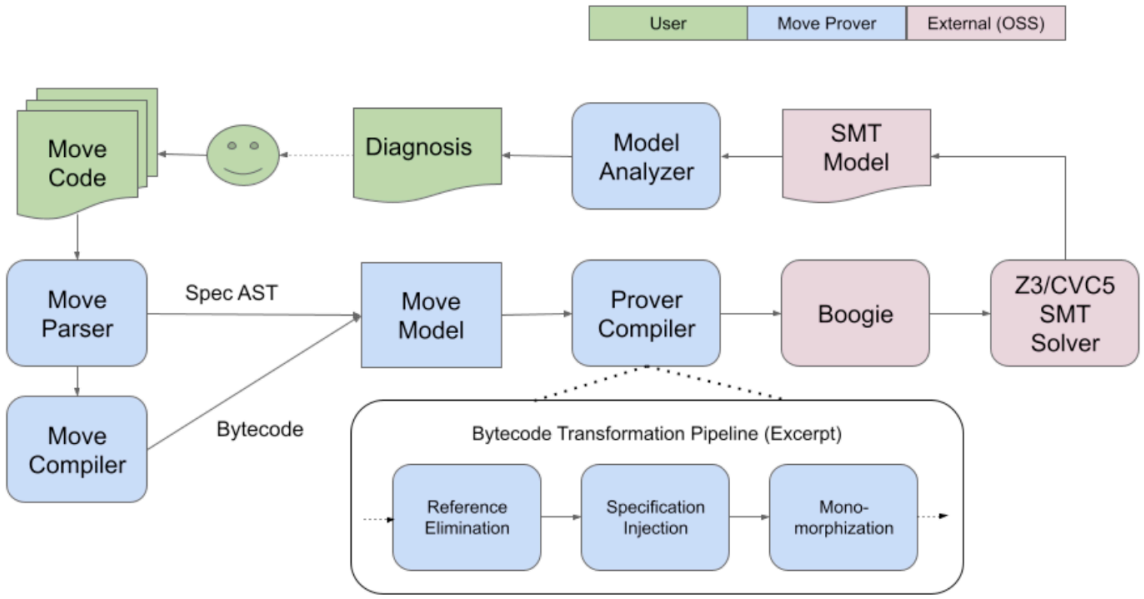


Figure 2.2: Move Prover Architecture, Source: Novi Research

Boogie (developed at Microsoft Research) is an intermediate language for verification between program (source code or bytecode) and SMT solver [29]. It is an abstraction layer for various theorem provers.

What the move prover does is implement each bytecode instruction as a Boogie procedure. A translated bytecode program then looks like a sequence of procedure calls. The specifications are also translated into boogie code + boogie assumptions and assertions.

The specifications are written in a subset of the Move language called Move Specification Language (MSL) [30]. These MSL specifications are then statically and exhaustively proven by the Move prover.

To give an example, MSL enables to write specifications, such as post-conditions of Move functions, which are then proven by the Move prover:

If the Move function is the increment function in Listing 2.1, then the post-condition specification can be the one in Listing 2.2.

Listing 2.1: Increment function in Move

```
fun increment(counter: &mut u64) { *counter = *counter + 1 }
```

Listing 2.2: Increment post-condition specification in Move

```
spec increment {
   ensures counter == old(counter) + 1;
}
```

As you can tell, the syntax is simple to understand, and thus enables the programmer to leverage formal verification at a high level.

Platform Agnosticism

The case with most new blockchains which do things a bit differently, is that they tend to come up with their own smart contract language. For example, Cardano came up with Plutus [18], Flow with Cadence [31], and Starknet with Cairo [32]. They do this because a lot of the implementation details of the underlying system are hardcoded, like the account, transaction, serialization format, choices of cryptography used and possibly also some of the consensus mechanisms. So if you want to adopt an existing programming language with an existing community, you will also inherit some of the design

decisions of the previous blockchain. The other option is to bootstrap an entirely new programming language including the community around it. Move keeps the language as simple and non-blockchain related as possible. All of these design concepts are not embedded in the Move language itself. These are put in at a higher layer. The lower layer is just Move, which does not know what is happening on the higher layer, which consists of all blockchain related concepts, such as the ones mentioned above.

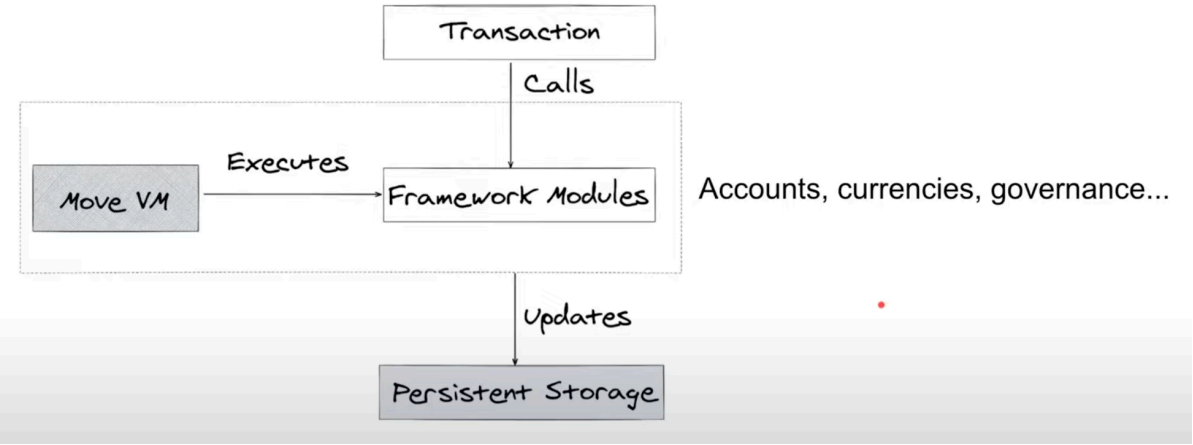


Figure 2.3: Move is Platform Agnostic, Source: Mysten Labs

In figure 2.3, everything except the VM can be customized.

Native Functions

Native functions are functions that can be used in Move modules, but do not have their implementation in the Move language itself. Their implementation can be found in the MoveVM, written in Rust. Usually these native functions are meant for standard library code, such as the rust vector standard library in Listing 2.3:

Listing 2.3: Empty vector native function in Move

```
module std::vector {
    native public fun empty<Element>(): vector<Element>;
    ...
}
```

The implementation of this function can be found in the MoveVM rust code in Listing 2.4:

Listing 2.4: native_empty rust implementation

```
pub fn native_empty(
    gas_params: &EmptyGasParameters,
    _context: &mut NativeContext,
    ty_args: Vec<Type>,
    args: VecDeque<Value>,
) -> PartialVMResult<NativeResult> {
    debug_assert!(ty_args.len() == 1);
    debug_assert!(args.is_empty());

    NativeResult::map_partial_vm_result_one(gas_params.base, Vector::empty(&
    ty_args[0]))
}

// This code can be found at: [https://github.com/move-language/move/blob/2412
    f877a5065132f31bfc339e6d1f2b9de10e87/language/move-stdlib/src/natives/vector.
    rs#L32-L42]
```

All the other standard native functions can be found in the code specifically at `https://github.com/move-language/move/blob/3ef3f1f3e18ef991a0f2790a60dc7bb47e6dae49/language/move-stdlib/src/natives/mod.rs#L105`.

It is also possible to create your own custom native functions, which can be added to the MoveVM easily by passing the natives on the MoveVM instantiation, as seen in Listing 2.5.

Listing 2.5: Adding custom native functions

```
pub fn new(
        natives: impl IntoIterator<Item = (AccountAddress, Identifier, Identifier, NativeFunction)>,
    ) -> VMResult<Self> {
        Ok(Self {
            runtime: VMRuntime::new(natives).map_err(|err| err.finish(Location::Undefined))?,
        })
    }

// This code can be found at:[https://github.com/diem/move/blob/725168d7522a3abeeb8664b4f1498552b3657286/language/move-vm/runtime/src/move_vm.rs#L24]
```

Transaction Scripts

MoveVM has the ability to natively invoke multiple already-published module procedures in one transaction. These procedures are written in Scripts. A Script is limited in the sense that it can only contain one main function, and in that main function it is only able to able to call functions of already deployed modules and cannot return a value. Scripts have very limited power—they cannot declare struct types or access global storage. Their primary purpose is to invoke module functions. An example of a simple script is shown in Listing 2.6.

Listing 2.6: Simple Script in Move

```
script {
   use 0x1::Math;
   use std::debug;

   fun main(a: u64, b: u64) {
      let sum = Math::add(a, b);
      debug::print(&sum)
   }
}
```

This script is then able to be executed in a transaction. Scripts cannot be reused as they are not published. This is a key difference between scripts and modules. Scripts are executed only once in a transaction. As a result, applications are safer, the user experience is improved, and there is greatly more flexibility.

2.1.3. Advantage in Preventing Mistakes

With all the safety-focused features Move has, it has strongly positioned itself as a safe smart contract programming language that is able to prevent many programming mistakes. Bugs in smart contracts can have devastating financial consequences, as can be seen by the total amount of value stolen over the years visualized in figure 1.5 and 1.6 of the introductory chapter.

Move could have prevented many of these hacks, as many bugs that caused these hacks are simply not possible if the smart contracts were written in Move. For example, the most researched vulnerability being the re-entrancy attack [26], where a function is called repeatedly in the same transaction potentially draining the smart contract from all its funds, is not possible in Move due to the resource model and explicit borrow semantics. Once a resource is moved, it cannot be moved again within the same call.

2.1.4. Move Syntax

Basic Structure

The syntax of a Move program defines how Move code is written and organized. Move programs are composed of packages and modules, with the latter containing the Move structs and functions. One package could contain multiple move modules. A typical Move program directory is organized as in figure 2.4.

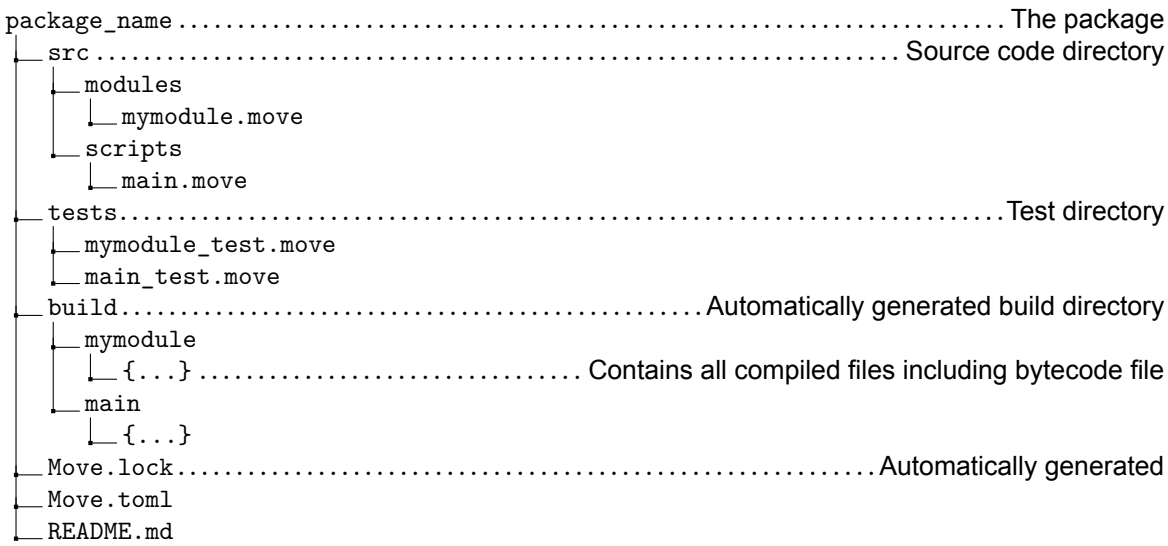


Figure 2.4: Directory Structure of Move Program

Basic Elements

A simple Move module in which a Coin with a certain value is being 'minted' to the function caller can be seen in listing 2.7.

Listing 2.7: Simple Move Module

```
module coin_address::user_coin {
    struct Coin has key {
        value: u64,
    }

    public fun mint(account: signer, value: u64) {
        move_to(&account, Coin { value })
    }
}
```

The Move.toml contains the package manifest, and has the following syntax in listing 2.8, with an '*' character representing optional fields and an '+' character representing one or more elements. A bare minimum implementation of Move.toml is given in listing 2.9.

Listing 2.8: Move.toml Syntax

```
[package]
name = <string>                 # e.g., "MoveStdlib"
version = "<uint>.<uint>.<uint>" # e.g., "0.1.1"
license* = <string>             # e.g., "MIT", "GPL", "Apache 2.0"
authors* = [<string>]           # e.g., ["Joe Smith", "Jane Smith"]

[addresses]  # (Optional section) Declares named addresses in this package and
    instantiates named addresses in the package graph
# One or more lines declaring named addresses in the following format
```

```
<addr_name> = "_" | "<hex_address>" # e.g., Std = "_" or Addr = "0xC0FFEECAFE"

[dependencies] # (Optional section) Paths to dependencies and instantiations or renamings of named addresses from each dependency
# One or more lines declaring dependencies in the following format
<string> = { local = <string>, addr_subst* = { (<string> = (<string> | "<hex_address>"))+ } }

[dev-addresses] # (Optional section) Same as [addresses] section, but only included in "dev" and "test" modes
# One or more lines declaring dev named addresses in the following format
<addr_name> = "_" | "<hex_address>" # e.g., Std = "_" or Addr = "0xC0FFEECAFE"

[dev-dependencies] # (Optional section) Same as [dependencies] section, but only included in "dev" and "test" modes
# One or more lines declaring dev dependencies in the following format
<string> = { local = <string>, addr_subst* = { (<string> = (<string> | <address>)) + } }
```

Listing 2.9: Bare minimum Move.toml file

```
[package]
name = "AName"
version = "0.0.0"
```

2.1.5. Architecture of MoveVM

When discussing Move, it is usually distinguished between two parts: the Move Language and the Move VM. The full Move language stack actually consists of four parts, as seen in the language directory of the Move Github repository [33]. This is the blockchain-agnostic part of Move that can be used to enable Move execution in a blockchain. As will be seen in the next chapter, this is the ”Move” that is being used by the Move blockchains. It is the engine that makes the chain move.

1. The Virtual Machine
    - Contains the bytecode format, a bytecode interpreter, and infrastructure for executing a block of transactions. This directory also contains the infrastructure to generate the genesis block.
2. The Bytecode Verifier
    - Contains a static analysis tool for rejecting invalid Move bytecode. The virtual machine runs the bytecode verifier on any new Move code it encounters before executing it. The compiler runs the bytecode verifier on its output and surfaces the errors to the programmer.
3. The Move Compiler
    - Contains the Move source language compiler.
4. The Standard Library
    - The standard library is a repository of default modules and native functions that are already developed and can be used by module developers. These are all the modules that you can import natively from ‘std‘. Examples are std::vector, str::string and std::debug.

These core components together form Move, as is visualized in figure 2.5. To publish a move package containing move source code, this move source code first has to go through a compiler, which produces the move bytecode. This bytecode is then verified by the bytecode verifier before it is being published. Being published means the code now resides on the chain. A transaction is able to invoke a state change in the global storage (the blockchain). This is done by the Move VM. The Move VM reads the required data for the transaction from the object store and executes this. It does so by loading all the necessary modules and dependencies for the execution with the help of the Loader found in the repo at `move-vm/runtime/loader.rs`, caches and verifies them as well. After successful execution of the

transaction, the Transaction Effects are produced. These effects are used to create the next state of the blockchain.

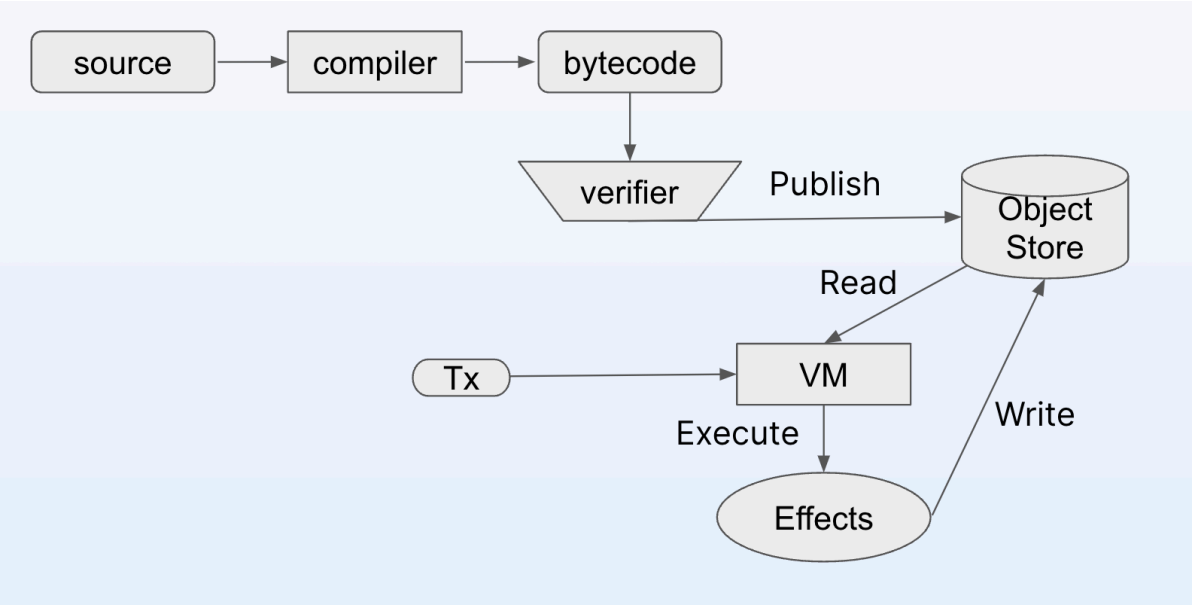


Figure 2.5: Move Compile/Publish/Run Toolchain, Source: Mysten Labs

Bytecode instructions

Move has a set of bytecode instructions which are typed. This means that the type of the instructions is retained, thus improving readability as it is more descriptive and closer to the high level code than untyped bytecode. These bytecode instructions can be found in the code here: `https://github.com/move-language/move/blob/main/language/move-vm/runtime/src/interpreter.rs`

Move source language can be disassembled into Move bytecode with the move disassembler CLI tool: move disassemble /path/to/move/source/file

The full bytecode of the usercoin module in Listing 2.7 is seen here in Listing 2.10:

Listing 2.10: Bytecode of User Coin Module

```
module 42.user_coin {
struct Coin has key {
        value: u64
}

public mint(account: signer, value: u64) {
B0:
        0: ImmBorrowLoc[0](account: signer)
        1: MoveLoc[1](value: u64)
        2: Pack[0](Coin)
        3: MoveTo[0](Coin)
        4: Ret
}
}
```

Bytecodes are variable size instructions for the Move VM. Bytecodes are composed by opcodes (1 byte) followed by a possible payload which depends on the specific opcode and specified in ”()” in the bytecode table found in Appendix A.

2.2. Aptos and Sui Flavors

2.2.1. Introduction

The Move discussed in the previous chapter actually has two major flavors: Aptos Move and Sui Move. These are the two biggest players in the scene and have emerged directly from the original Diem Move team. This chapter will focus only on the largest two projects, Aptos and Sui, due to their continuous contribution to the MoveVM and ongoing large-scale research efforts.

Both teams consist of previous Facebook employees that worked on the Diem blockchain. Since the downfall of the Diem project, these two teams have continued the development of Move in their own separate blockchains. Initially, this was a joint approach/collaboration as can be seen on the original Move github page [34]. Both Aptos and Sui worked on their flavor of Move in separate branches. Later on, both decided to continue on their own fork and leave this joint collab alone.

Aptos released their blockchain in mainnet on October 17th 2022, while Sui reached mainnet May 3rd 2023.

As seen in the previous section of this chapter, Move is platform agnostic. This means that it is possible to adopt Move in a different DLT, without having the burden of having to customize much of the VM.

Aptos Move basically uses Diem textbook Move, utilizing the commonly known address-centric or account-based global storage. Sui uses a different object-centric model, in which elements such as assets and modules on the blockchain are represented via objects.

Each deployment of the MoveVM has the ability to extend the core MoveVM with additional features via an adapter layer. Furthermore, MoveVM has a framework to support standard operations much like a computer has an operating system.

2.2.2. Overview of Aptos Move and Sui Move

This will be an exploration of key features and functionalities of both flavors. A high level non-exhaustive comparison between Aptos Move and Sui Move can be seen in table 2.1.

Attribute	Aptos Move	Sui Move
Data storage	Stored at a global address or within the owner's account	Stored at a global address
Parallelization	Capable of inferring parallelization at runtime within Aptos	Requires specifying all data accessed
Transaction safety	Sequence number	Transaction uniqueness
Type safety	Module structs and generics	Module structs and generics
Function calling	Static dispatch	Static dispatch
Authenticated storage	Yes	No
Object accessibility	Guaranteed to be globally accessible	Can be hidden

Table 2.1: High level comparison Aptos/Sui Move.

Storage Model

The main difference between Aptos and Sui is the storage model, or the state of the ledger. This difference can be described using the physical memory analogy: *Aptos uses unified memory, while Sui uses partitioned memory.*

Physical memory is a large array of bytes, and each byte (word) has an address through which it can be accessed. Multiple programs (transactions) try to use the memory simultaneously, possibly resulting in concurrency conflicts. The operating system (consensus engine) needs to decide which program can use which memory location concurrently.

There are two main approaches to this:

1. **Account Based:** Each program has exclusive access to the whole memory. This is the approach used by Ethereum.

2. **UTXO Based:** Each program has exclusive access to a subset of the memory, and it needs to declare upfront which memory locations it wants to use. This is the approach used by Bitcoin.

Conflicts are more easily detected in the UTXO based case, as the operating system knows which memory locations are used by which program. In the Account based case, conflicts are resolved by sequential execution, one program after the other.

The UTXO approach is a simple and beautiful when it is looked at from the perspective of the protocol developer. While this is the case, it is not very suitable for complex applications that require a shared state. Imagine a shared state, i.e. a DEX pool, is represented with a UTXO. All user transactions interacting with this DEX pool commit to a certain state (the UTXO to spend), and there can only be one winning transaction. All other transactions are discarded as conflicts.

So how that plays out as a user, is that the user needs to submit the trade, wait to see if that trade is won, and if not, resign and submit the trade again. This is not a great user experience, as the user will not know how many times the trade has to be resubmitted until it has won. This also wastes the network bandwidth with failed transactions.

In the case of the account based ledger, the user submits the trade and waits for the consensus engine to order the transaction. The user does not have to commit to a certain state to trade against, but it will eventually happen. The downside is that the user has to deal with transactions that are not deterministic. It is unclear how exactly the trade will happen, as the consensus engine can order this transaction in any way that it wants. This is the reason that most DEX swap protocols require you to set a maximum slippage to protect against trading against a price that is drastically different than you expected to trade against.

An object based ledger is a middle ground between the previously mentioned two extremes. It uses the memory access list of the UTXO approach, but allows for shared state to exist at specific memory locations. This way, only programs that compete for the same memory location need to be ordered. If programs compete for different memory locations, they do not need to be ordered.

Aptos makes use of an account-based global storage as visualized in 2.6, which has similarities to most blockchains. This model is identical to the one used in the original Diem code. Ethereum for example uses an account-based model as well, where the global state consists of addresses which contain the data. This ledger is simply a key-value store with addresses as keys and resources as values. This model has a few notable drawbacks:

1. It assumes that all transactions are totally ordered, as all transactions have access to the same total global state. Each transaction updates the complete global state, after which the next transaction uses that newly updated state.
2. Transactions are grouped into blocks which are executed in batches. Validators pick and order transactions based on their gas fees to maximise profits. This allows for MEV opportunities such as front-running and sandwich attacks.
3. Since transaction execution updates the full global state, the cryptographic root hash of the state must be calculated too. Calculating this hash is computationally expensive.
4. Parallelization of transaction execution is not an obvious ”thing”, because each transaction reads and writes the same global state. However, it is possible and Aptos made it happen with their Block-STM approach: [https://arxiv.org/pdf/2203.06871.pdf], an in-memory parallel execution engine.

Ledger activity (due to transactions) consists mostly of state transitions which changes data associated with addresses. This means that each transaction will invoke two ledger updates: one update on the sender address, and one update on the receiver address.

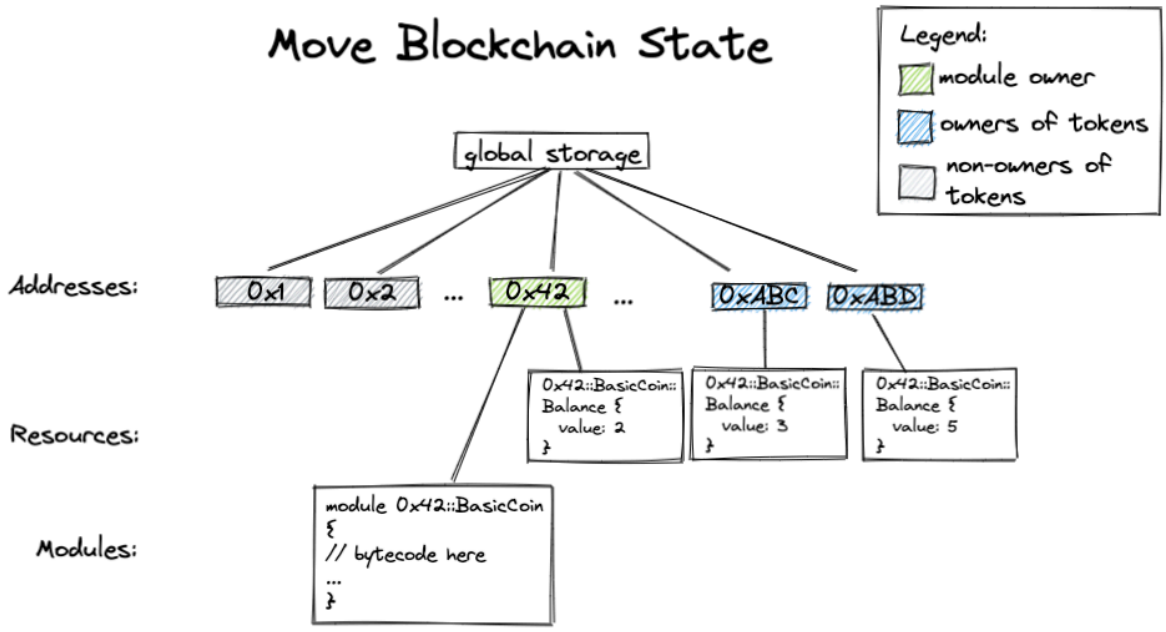


Figure 2.6: Account-based model of Aptos. Source: Cetrik

Sui makes use of an object-based global storage, which has similarities with UTXO models like Bitcoin, but also some aspects of account-based models. In UTXO models, every UTXO is owned by one owner, and only the owner is able to use this UTXO. In Sui's model, you have two types of objects: shared and owned objects. Owned objects have the same characteristics as UTXOs in the sense that they can only be owned by one owner. This inherits the same benefits as UTXOs, namely that they can be executed in parallel.

In the Sui model, ledger activity from transactions does not consist of two changes in addresses, but only one: the object. In the case of a simple transfer of an asset between two participants, the only change required is the one to the "owner" label of the object.

Consensus

Aptos uses AptosBFT, which is basically LibraBFT. It makes use of Byzantine Fault Tolerance (BFT) and Proof of Stake (PoS).

Sui uses Narwhal and Bullshark. Narwhal is the mempool engine and Bullshark is the consensus engine. The mempool has the task of delivering the required data to the consensus engine, while the consensus has the task of agreeing on a specific order of that data.

Move Flavor

The Aptos Move adapter features include the following [35, 36]:

- Move Objects that offer an extensible programming model for globally access to heterogeneous set of resources stored at a single address on-chain.
- Resource accounts that offer programmable accounts on-chain, which can be useful for DAOs (decentralized autonomous organizations), shared accounts, or building complex applications on-chain.
- Tables for storing key, value data within an account at scale.
- Parallelism via Block-STM that enables concurrent execution of transactions without any input from the user.
- Cryptography primitives, which include cryptographic hash functions (such as SHA2-256, SHA3-256, Keccak256, and Blake2b-256), digital signature verification algorithms (such as Ed25519,

ECDSA, and BLS), elliptic curve arithmetic (supporting curves like Ristretto255 and BLS12-381), and zero-knowledge proofs (such as Groth16 ZKP verification and Bulletproofs ZK range proof verification)

Sui Move differs from other Move versions [37]:

1. Sui uses its own object-centric global storage
2. Addresses represent Object IDs
3. Sui objects have globally unique IDs
4. Sui has module initializers (init)
5. Sui entry points take object references as input

Parallelization

Aptos parallelizes transactions through Block-STM [38]. Simply put, this method makes use of an optimistic parallel execution approach, where the transactions are first run in parallel and only after execution they will be validated. If anything goes wrong, the transaction can be aborted or re-executed [39].

Sui is able to parallelize transactions due to its object-based and ownership model. Transactions concerning owned objects can be executed in parallel in case there are no shared dependencies, and are able to skip consensus since there is no ordering necessary. Shared objects logically do have to go through consensus and can thus not be parallelized.

Move Code

The differences between these two flavors of Move has an impact on the Move module code that a developer has to write. This can be most prominently seen in the different way of thinking that is required for the different storage models: a Sui Move developer needs to think in objects, while an Aptos Move developer has to think in accounts that contain these objects. For example, in a Sui Move module, when you create a function that requires multiple different objects as input, even though they belong to the same user, you still have to get their individual references as arguments of the function. In Aptos Move, you do not have to provide these objects as arguments, as they are user-owned, and therefore reside in the user address as a struct. In the function, you simply check if the user address contains that specific struct, and access it that way.

3

Applying MoveVM to the IOTA DLT

3.1. Introducing IOTA

IOTA is an open, feeless data and value transfer protocol that utilizes a DAG-structured DLT called the Tangle [2].

3.1.1. Overview of IOTA and the Tangle

In the IOTA protocol, new transactions validate two previous ones with a small proof-of-work, ensuring scalability and decentralization. This means that theoretically, the more transactions occur, the faster the chain can process new transactions. To illustrate how the Tangle works, consider the following example:

Alice wants to send funds. She receives two previous transactions from Bob and Sarah as potential candidates (”tips”) for verification, as they made their transactions just before her. To prevent issues like ”double spending,” Alice’s computer verifies both transactions. After performing a small proof-of-work, her transaction is added to the IOTA network. A supervisor (the ”coordinator”) then reviews these verified transactions and, after a final check, broadcasts confirmation to the network through a ”milestone transaction.”

3.1.2. IOTA 2.0

IOTA comprises two main networks: the IOTA mainnet and the Shimmer staging network, used for testing and deploying updates to the protocol. After validation, these updates can be deployed to the IOTA mainnet.

IOTA 2.0, currently in development with a public testnet released in mid-2024, aims to remove the coordinator and introduce MANA, a secondary token used for gas fees. This MANA token is passively generated by IOTA holders and also awarded for delegation and validation, preserving the original ”feeless” aspect of the DLT. The lifecycle of MANA is visualized in Figure 3.1.

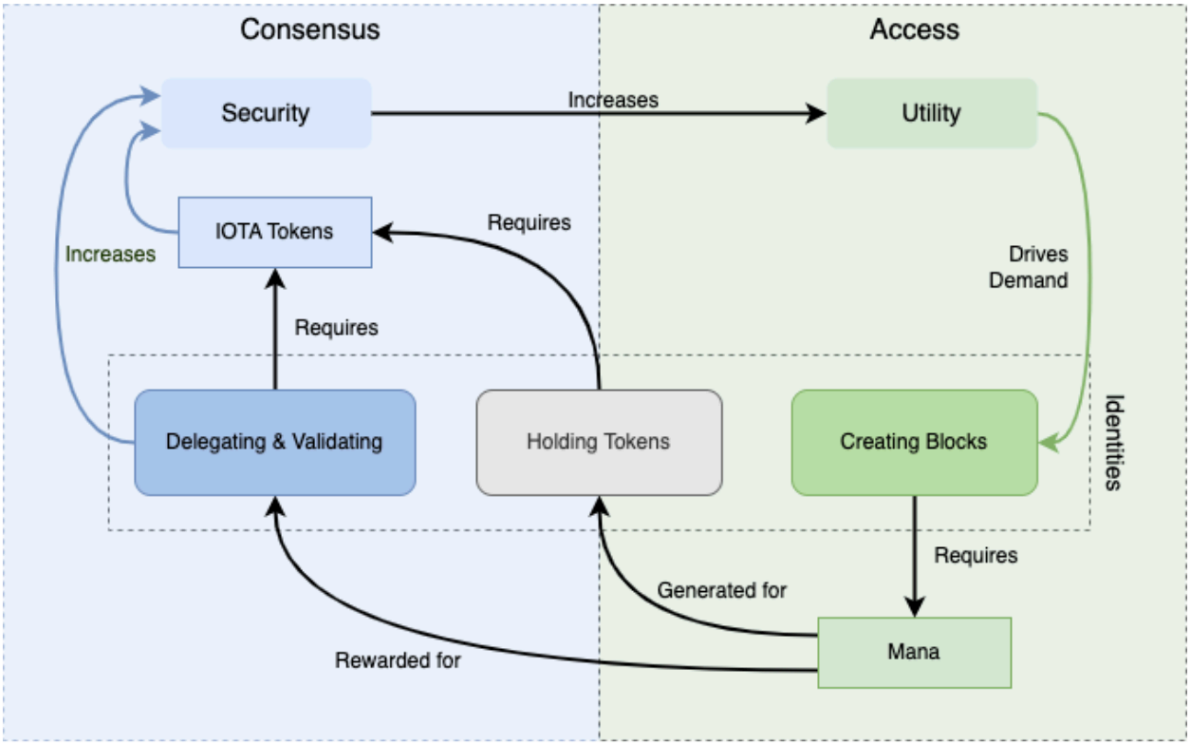


Figure 3.1: MANA lifecycle, Source: IOTA Foundation

In IOTA 2.0, transactions are executed on arrival in parallel, utilizing a Parallel Reality-Based Ledger [40]. Conflicts between transactions and their dependencies are tracked in a conflict DAG and resolved there.

Understanding the life cycle of a transaction is crucial to appreciating the differences between traditional blockchains and IOTA's Tangle. In a typical blockchain, transactions go through several stages including broadcasting, validation, and inclusion in a block by operators. This process can involve significant delays and fees due to the need for sequential processing and competition among miners.

In contrast, IOTA's Tangle operates differently. Transactions are validated through a process where each new transaction confirms two previous transactions, creating a web of interlinked transactions. This method allows for parallel processing, reducing delays and eliminating fees.

Table 3.1 provides a step-by-step comparison of the transaction life cycle in traditional blockchains versus IOTA.

Table 3.1: Comparison of Steps in a Typical Blockchain vs. IOTA

Typical Blockchain	**IOTA**
1. Transaction created and gossiped to the network	1. Transaction packaged into a block and gossiped
2. Transaction sits in the mempool	2. Transaction executed on arrival
3. Transaction sequenced into a block	3. Block receives approvers
4. Block issued to the network	
5. Block is validated	
6. Transaction executed	
7. Block received approvers	

3.1.3. Consensus Mechanism

The current IOTA protocol, called Stardust [41], uses the Coordinator. The Coordinator emits milestone blocks that nodes trust and use to confirm blocks. Blocks referenced by these milestone blocks are considered confirmed. This milestone block acts similarly to a block header in traditional blockchains, but in the case of IOTA, it also confirms a subgraph of the DAG. The Coordinator operates on the Tendermint Core BFT consensus, allowing a committee of validators to function as a distributed Coordinator.

In IOTA 2.0, consensus is reached through an adaptation of Nakamoto Consensus applied to a DAG known as the Tangle. Unlike traditional blockchains, the Tangle's architecture supports a dynamic network of linked blocks. Each node contributes to consensus by validating transactions and blocks. Nodes build the Tangle by linking to earlier blocks, guided by a random tip selection algorithm for block inclusion. Consensus flags and slot commitment chains maintain consensus across nodes, switching to the heaviest sub-chain during network issues. This ensures a unified, consistent ledger view for all participants, enhancing the security and efficiency of the IOTA network.

3.1.4. Smart Contract Capabilities

IOTA 2.0 does not have native L1 smart contract capabilities. Instead, IOTA Smart Contracts (ISC) allows smart contract blockchains to connect to the IOTA Tangle. This means multiple chains with smart contract capabilities can connect to the IOTA Tangle, executing in parallel for higher throughput and lower fees. ShimmerEVM is an example, running as a network with the Ethereum Virtual Machine (EVM) on top of the IOTA network. Although this approach enables smart contracts on IOTA, it relies on Layer 2 (L2) solutions, which introduce trust in L2 coordinators.

The absence of L1 programmability in IOTA limits its flexibility. L1 programmability would enhance the network's economic security by increasing the value of MANA, as well as its usage and demand. Additionally, L1 programmability provides essential building blocks for zero-knowledge technology, name services, rollups, sidechains, and other potential L2 solutions. It would also prepare the L1 for future technological advancements.

3.1.5. Architecture Parallels between IOTA and Sui

Understanding the core principles and architectures of both Move flavors reveals some obvious similarities between Sui and IOTA, namely the object-centric storage model (UTXOs in IOTA, Objects in Sui) and parallelization of transactions. Neither IOTA nor Sui have a global account-based ledger like Ethereum or Aptos.

IOTA aims for the parallelization of transactions. This is similar to what Sui aims to do with its UTXO-object-based ledger.

The IOTA protocol mainly consists of the following primary layers:

1. **Networking:** Manages discovery and selection of neighbors and their connections so that information can be efficiently exchanged. It also manages the gossip between nodes and their peers.
2. **Data Structures/Communication Layer:** The Tangle is a block DAG where each block has a size of just 1. Each block can reference up to 8 parent blocks. IOTA 2.0 makes use of the Unspent Transaction Output (UTXO) model. Write access is simultaneously permitted in parallel by multiple participants, reflecting natural causal order. A scheduler limits write access using MANA as sybil protection. This is part of the communication layer.
3. **Tangle 2.0 Consensus:** The protocol uses Tangle 2.0 consensus: Leaderless Nakamoto Consensus on the Heaviest DAG.
4. **Staking and Mana Incentives:** MANA is the essential resource that binds all components of the protocol together. Participants are rewarded with system access for engaging in consensus.

A Block: A block is the basic unit of information, essentially a wrapper or container for all data. Each block contains the following information:

- Timestamp
- List of parents

- Payload (transaction or data)
- Slot commitment: cryptographic summary of a slot
- Amount of MANA burned
- Issuer Account
- Issuer Signature

These blocks are categorized based on time slots. The timeline is divided into segments called slots, each uniquely indexed. The slot a block belongs to can be determined based on its timestamp.

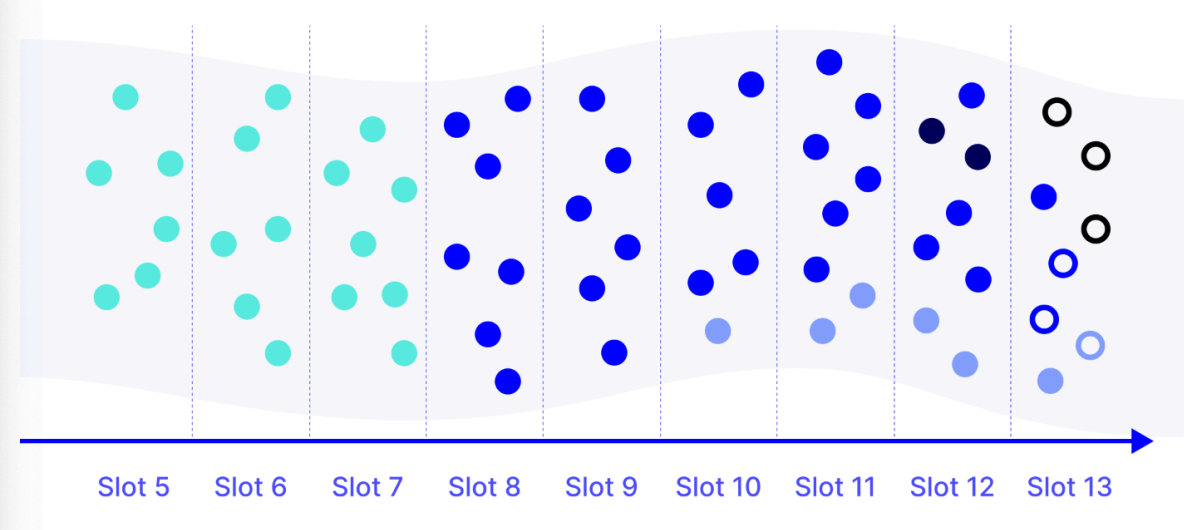


Figure 3.2: IOTA Time slot, Source: IOTA Foundation

Payloads are packages of data contained within each IOTA block, except for validation blocks, which do not contain any payload. Each payload consists of a header that specifies the payload type. IOTA 2.0 contains two types of payloads:

- **Transactions:** Payloads used to transfer value. Contains two fields: header and Transaction Essence. The latter includes all transaction data, such as Network ID, Creation Slot Index, UTXO Input List, UTXO Output List, and Extra Payload.
- **Tagged Data:** Generic data payload with a customizable tag field. Contains three fields: header, tag, and data.

3.1.6. MoveVM for IOTA L1 Programmability

MoveVM's blockchain agnosticism suggests that achieving L1 programmability on IOTA could be possible, and the Sui Move flavor seems to be the most compatible. The practical implications and implementation of such a design are documented in the following chapters. This process is split into two phases: the first phase focuses solely on achieving programmability with MoveVM, without actual real node software. The second phase continues the work of phase 1 by implementing the real node software and consensus.

3.2. Phase 1 - The Adapter Prototype

3.2.1. Introduction

Making MoveVM work with IOTA requires creating an adapter for it. In essence, MoveVM is just a black box. Bytes can enter the box, and it outputs a result. This result is a set of instructions that can be used to modify the state, called a `TransactionEffect`. The black box is something that should not be touched or modified. It was specifically designed this way so that MoveVM could be blockchain agnostic. Tooling needs to be built around it to connect it to the rest of the IOTA DLT. The black box

does not know of concepts like accounts, storage, addresses, transactions, etc. These need to be defined within what is called the adapter and framework.

This section is focused mainly on the adapter, and is denominated with phase 1 of the project.

3.2.2. Prototype Architecture

The MoveVM is exposed by a simple API interface, which just outputs `TransactionEffects`, or a `WriteSet`. Building the necessary adapter and blockchain concepts around it results in a high level architecture like the one shown in figure 3.3.

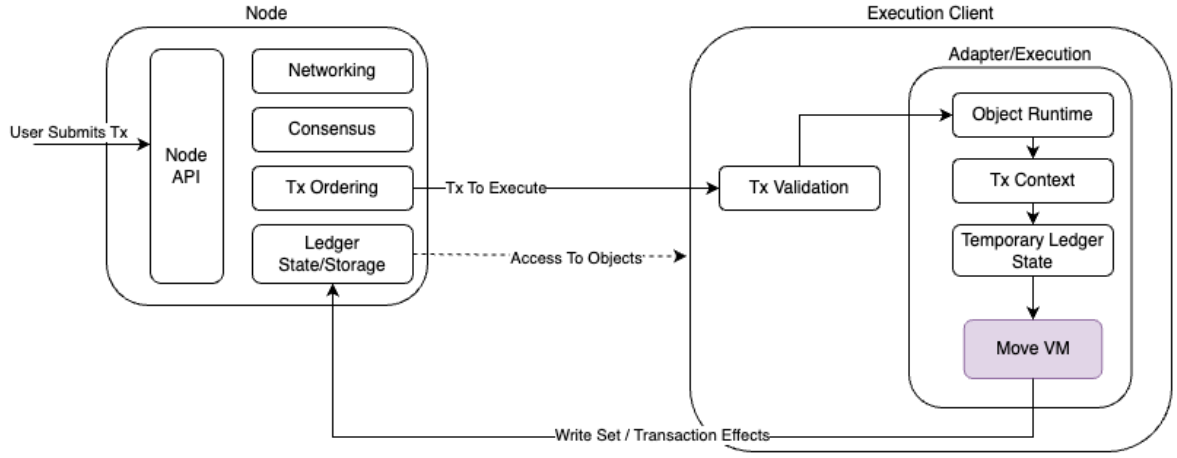


Figure 3.3: High Level Adapter Architecture

In this architecture, the node's function is to maintain the ledger state, run the consensus engine and network with peers. The consensus engine runs the consensus algorithm which determines which transactions need to be executed. In phase 1, a simplified version is used where consensus does not play a role, as the nodes treat the Move transactions as simple binary payloads. This is the reason consensus is not able to validate them, and the reason why a separate validation step is included in the Move Adapter. Once a transaction passes the consensus engine, it is passed to the Execution Client to be executed. This is handled by the Move Adapter and Move VM.

3.2.3. Phase 1 Simplification

As consensus is not necessary for this phase of the project, which focuses on getting MoveVM to work, the decision was made to simplify the node implementation by not running the consensus algorithm and networking. The dummy node will only maintain the ledger state, which is a collection of objects. This node can accept incoming transactions and execute them sequentially in the current ledger state. In this prototype, there will be no Tangle and its blocks. A user simply submits raw move transactions to the node.

3.2.4. Transactions

Transaction Processing

Validation Phase The adapter has the task of executing a transaction and producing a writeset which can be used to apply changes to the ledger state. However, executing any arbitrary payload is not possible. This payload, which is the transaction, first needs to be validated before it enters the black box, which is the MoveVM.

The validation phase consists of the following tasks, in arbitrary order:

- Ensuring the payload is well-formed and syntactically correct.
- Ensure the signatures are valid.

- Ensure that the referenced objects exist, with correct version ID and matching digests in case of owned/immutable objects.
- Check if the accessed objects are accessible by the sender of the transaction.
- Check if the transaction fields respect the protocol rules, such as adhering to the max number of arguments.
- Check if gas payment objects are sufficient to cover the maximum budget of the transaction.

If any of these conditions fail, then the transaction will not pass the validation phase and will be considered an `Illegal` transaction. Such illegal transaction is not executed and does not produce a writeset and thus no change in the ledger is made.

All these validation checks happen before the execution starts, and could therefore technically be done by the node itself. It is not necessary for this logic to be incorporated in the Execution Client, as it does not need anything from the Move Adapter. It might even be better to incorporate this logic in the Node, but for the sake of simplicity in this phase, this is done in the Execution Client.

Execution Phase After the validation phase comes the execution phase. This is where the adapter processes the transaction and creates a transaction context. It then creates a temporary ledger state out of all the objects in the execution context, but retains an online link to the node's ledger state so that it can always fetch additional objects on-demand, which is used to facilitate Dynamic (Object) Fields [42].

Execution will always produce a write-set that is committed to the ledger. This write-set could be one in which the transaction is executed successfully, or one in which the transaction has failed and is aborted. In the latter case, there will be no changes to the ledger state except for the transaction in which the gas payment is deducted from the sender. This deduction is always committed to the ledger whenever there is a submitted transaction.

Transaction Structure

The actual structure of a Move transaction is made of two parts:

- Intent message, which contains the transaction payload. In IOTA, this is called the `TransactionEssence`.
- List of signatures, which is a vector of all signatures of the senders. In our prototype, only one signature with Ed25519 signature system will be supported.

Listing 3.1: SenderSignedTransaction struct

```
pub struct SenderSignedTransaction {
    /// The unsigned transaction data.
    pub intent_message: IntentMessage<TransactionData>,
    /// The signature for the transaction data.
    pub tx_signature: GenericSignature,
}
```

The `TransactionData` that can be seen in the `SenderSignedTransaction` struct in listing 3.1 is another struct which contains the operation that the transaction wishes to perform with some additional metadata. This `TransactionData` is shown in the Listing 3.2 below.

Listing 3.2: TransactionDataV1 struct

```
pub struct TransactionDataV1 {
    /// The transaction kind. Defines the operation(s) to carry out.
    pub kind: TransactionKind,
    /// The sender of the transaction.
    pub sender: IotaAddress,
    /// Gas payment information (gas payment objects, gas budget, etc.)
    pub gas_data: GasData,
}
```

In phase 1 prototype, two types of transactions are supported:

- Transaction to publish a Move module.
- Transaction to Call a Move Function.

The implementation of this enum is shown below in Listing 3.3.

Listing 3.3: TransactionKind enum

```
pub enum TransactionKind {
    MoveCall(MoveCallTransactionData),
    Publish(MoveModulePublish),
}
```

Publish Transaction

The publish transaction is a special type of transaction that enables the user to publish a new Move module to the ledger. This transaction contains the compiled Move bytecode of the to be published Move package, which is simply a list of modules. The struct of this specific transaction type is shown in Listing 3.4 below.

Listing 3.4: MoveModulePublish struct

```
pub struct MoveModulePublish {
    pub modules: Vec<Vec<u8>>,
}
```

The Move compiler generates these bytes from the package source code using `.toml` package file. It compiles all `.move` source files into move bytecode and adds some metadata and type information. It then finalizes it by serializing the result into a binary format with Binary Canonical Serialization [43].

After execution of this publish transaction, a new package object is created in the ledger state with the bytes of the compiled modules into the package's modules field.

Call Transaction

When a module is published, it exposes its functions. These can be called by so called Call Transactions. One of these is the transfer function which is a crucial function of a ledger. A Move call transaction needs to contain the following information:

- Which function of which module and package is called.
- What are the arguments to this function.
- If it is a generic function, which type arguments to use.

These can be seen in the `MoveCallTransactionData` struct in Listing 3.5 below.

Listing 3.5: MoveCallTransactionData struct

```
pub struct MoveCallTransactionData {
    /// The package containing the module and function.
    pub package: ObjectID,
    /// The specific module in the package containing the function.
    pub module: Identifier,
    /// The function to be called.
    pub function: Identifier,
    /// The type arguments to the function.
    pub type_arguments: Vec<TypeTag>,
    /// The arguments to the function.
    pub arguments: Vec<CallArg>,
}
```

The `CallArg` struct contains the arguments that are supplied to the Move function, which can be seen in Listing 3.6.

Listing 3.6: CallArg struct

```
pub enum CallArg {
    // contains no structs or objects
    Pure(Vec<u8>),
    // an object
    Object(ObjectArg),
}
```

Pure arguments are numbers, strings, boolean values, addresses, etc. Object arguments are references to objects in the ledger state. In these objects, a difference between the type of object is made. If they are owned objects, the `ObjectArg` struct is an `ImmOrOwnedObject` which contains `ObjectReferences`. These include `ObjectID`, `Version` and `Digest`.

If they are shared objects, the `ObjectArg` struct is an `SharedObject` which includes the `ObjectID`. The `ObjectArg` struct implementation can be found in Listing 3.7.

Listing 3.7: ObjectArg struct

```
pub enum ObjectArg {
    // A Move object, either immutable, or owned mutable.
    ImmOrOwnedObject(ObjectRef),
    // A Move object that's shared.
    // SharedObject::mutable controls whether caller asks for a mutable reference
    to shared
    // object.
    SharedObject {
        id: ObjectID,
        initial_shared_version: SequenceNumber,
        mutable: bool,
    },
}
```

3.2.5. Execution Flow

The overall execution flow of a Move transaction is depicted in figure 3.4. This is the simplified execution flow, in which only the most crucial elements are shown. A more detailled version of the execution flow can be found in Appendix C.

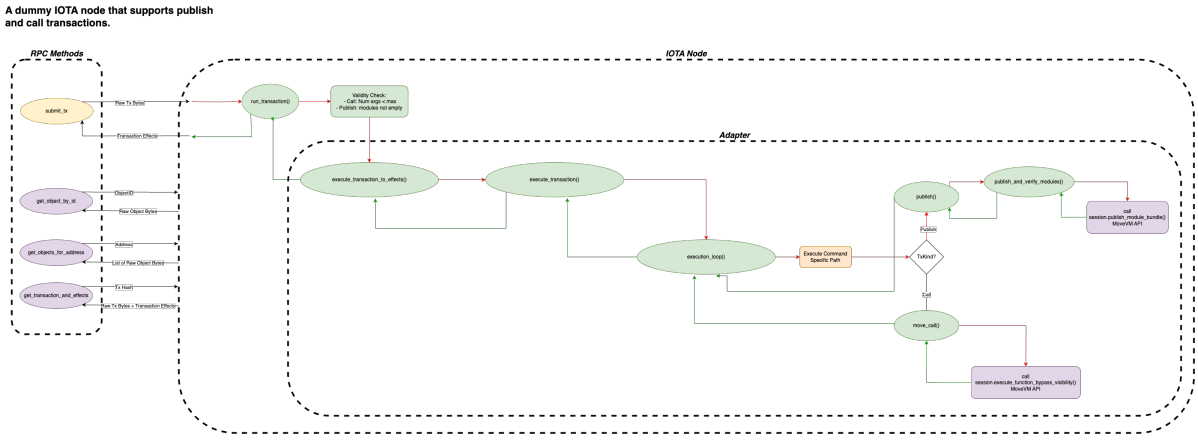


Figure 3.4: Simplified Execution flow of Move transaction in phase 1

In this figure, the red arrows outline the execution path from the node to the adapter, concluding in the MoveVM (purple boxes). On the other hand, green arrows signify the information flow from the VM back to the node, where the write-set (Transaction Effects) is ultimately committed to the ledger.

Notably, transaction validation occurs prior to the execution phase and is distinct from the adapter's functions. Upon successful validation, the adapter initiates the transaction context, starts gas metering, and executes the transaction based on its type.

In the following subsections two different transaction types will be explained in more detail.

Publish transaction execution

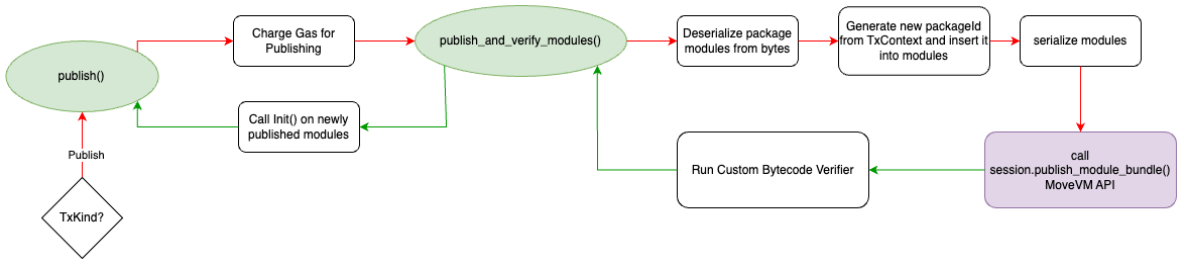


Figure 3.5: Execution flow of Publish transaction in phase 1

The general execution flow of a publish transaction is visualized in 3.5. The process begins by extracting the module bytes from the transaction and parsing them into a `CompiledModule` as defined by the Move language. Subsequently, a new package ID is generated for the package and incorporated into the `CompiledModule` as the `package_address`. These modules are then serialized back into bytes and forwarded to the MoveVM for publishing through its `publish_module_bundle()` API. At this stage, the MoveVM executes integrity checks on the bytecode, resolves dependencies by linking the modules, and upon successful completion, returns the on-chain module bytes of the package. Additionally, a custom bytecode verifier is employed to enforce IOTA-specific constraints on the package. Given that a Move dialect designed for object handling is used, the bytecode must adhere to this dialect's specifications, verified through static code analysis as part of the custom bytecode verifier process. It is important to note that packages include an init function that must be executed before saving the package, allowing for state initialization, creation of singleton objects, and other necessary actions.

Move call transaction execution

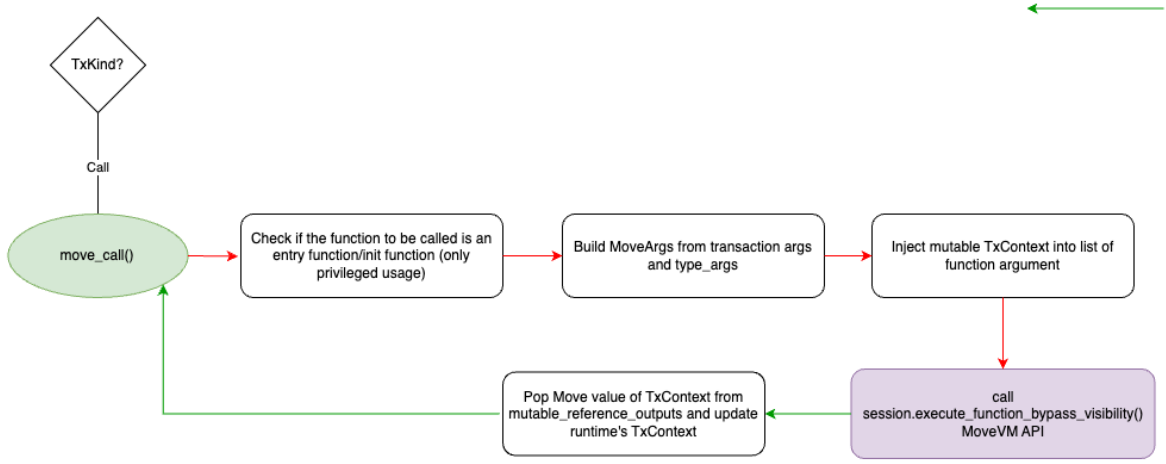


Figure 3.6: Execution flow of Call transaction in phase 1

Likewise, the general execution flow of a call transaction is visualized in 3.6. In the current iteration of our prototype, direct calls to entry functions are restricted when executing transactions, providing a simple interface. However, our upcoming Programmable Transaction feature in the production adapter will expand these capabilities. Despite this evolution, the primary role of the adapter remains consistent: it verifies that transaction arguments align with the called function's signature and

injects the Transaction Context into the function's argument list. This ensures the context's availability within MoveVM for on-chain code operations, such as retrieving the current time/epoch in Move code and generating new object IDs within the runtime. The actual function invocation occurs via the `execute_function_bypass_visibility()` API within MoveVM. Following MoveVM's execution, the adapter's runtime processes the result, preparing the transaction effects accordingly.

Transaction Effects

After a transaction is executed, it produces `TransactionEffects`. These contain the references to the write-set of the transaction. They also contain other information such as consumed gas and emitted events. The full list of variables in a `TransactionEffects` struct can be found in Appendix D.

3.3. Phase 2 - Hornet Prototype Architecture

3.3.1. Introduction

Phase 2 of the project focuses on combining the built adapter prototype with the actual working node software of IOTA. So instead of using a mocked node, the Hornet node of IOTA will be used. This phase is called the Hornet Prototype, and will focus on its architecture in this chapter. The next chapter will include much more detail in each of the individual components of the Hornet prototype. The diagram in Figure 3.7 gives an overview of all components of the prototype architecture.

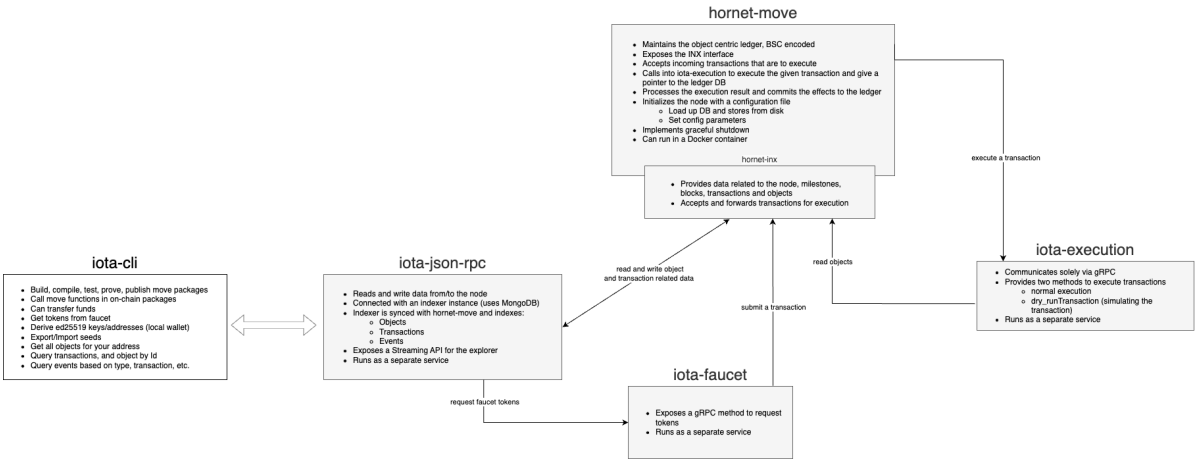


Figure 3.7: Prototype architecture

This figure contains the following components:

- **hornet-move:** The original Hornet node forked and modified to work with MoveVM and Objects instead of UTXOs.
- **iota-execution:** Responsible for transaction execution, contains the adapter.
- **move-inx:** Handles data related to node, milestones, blocks, transactions, and objects. It accepts transactions and forwards them for execution.
- **iota-json-rpc:** Provides a structured way to read, write, and stream object and transaction related data from the node.
- **iota-faucet:** Sends a transaction to the user with tokens so that the user can use it for gas payments when interacting with MoveVM.
- **iota-cli:** A command line interface tool which enables interactions with MoveVM and Hornet's object ledger.

The source code for all of these components is hosted in a private Github repository belonging to IOTA.

The objective is to demonstrate that the WhiteFlag algorithm can function as an effective sequencing algorithm that can sequence Move transactions and work with the results of MoveVM. Another objective is verifying the compatibility of WhiteFlag with `TransactionEffects` (created, mutated and deleted)

instead of only UTXOs (created and spent). In the end, different scenarios are tested in Move with WhiteFlag and thereby ensured that the sequencing functions accordingly and produces the right ledger state. With this result, it can be safely concluded that MoveVM has been successfully integrated with IOTA.

3.3.2. Prototype Components

hornet-move

The original Hornet node software is written in Go [44], and has been modified to make it work with Objects instead of UTXOs and MoveVM for smart contract execution. The modifications are summarized as follows:

- Swapped out the UTXO ledger state and database to facilitate object-centric operations.
- Swapped out the original transaction format with the new Move transaction format.
- Set up gRPC communication channel between Hornet and the execution engine written in Rust.
- Modified the WhiteFlag algorithm to enable Move transactions execution.
- Modified the ledger database to facilitate object management, including milestone cones, transaction application and rollback, pruning, and snapshotting.
- Refactored the INX modules and made some modifications to the coordinator and spammer modules.
- Testing with movement/fixtures testing suite.

Communication between Hornet and iota-execution The Hornet software written in Go needs to communicate with the execution software written in Rust. There are numerous ways of achieving such communication, such as direct bindings between Rust and Go by making use of libraries, or making use of a dedicated communication protocol such as gRPC, which has already made use of in the Phase 1 adapter prototype.

The first option could be achieved by making use of the UniFFI crate for Rust, which is a tool that automatically generates foreign-language bindings that target Rust libraries [45]. This tool generates an iota-execution go package which can be imported by Go.

UniFFI also has a convenient flag (UNIFFI_ENABLED) which can disable the bindings on the go, which makes it easy to compare the differences in execution time. This way, it is possible to benchmark the end-to-end time it needs to execute four transactions. The test commands used can be found in Listing 3.8:

Listing 3.8: UniFFI Flag testing execution times

```
UNIFFI_ENABLED=true go test -run ^TestScenarioPublishCallSimplePackage$ github.com
    /iotaledger/hornet-move/pkg/whiteflag/test -v

# iota_execution needs to be run for this test.
UNIFFI_ENABLED=false go test -run ^TestScenarioPublishCallSimplePackage$ github.
    com/iotaledger/hornet-move/pkg/whiteflag/test -v
```

This gave the following results, running the test 100 times:

- UniFFI: 55.898 seconds.
- gRPC: 55.223 seconds.

The actual output with the results can be found in Appendix E in Listing E.1.

From the results, it can be clearly seen that the execution times are similar. In fact, using gRPC is even a bit faster. Therefore, it is decided to go for a gRPC implementation as it brings multiple benefits compared to direct linking:

1. gRPC is capable of streaming requests and responses, which allows the client and/or server to stream data. In our usecase, constant communication between the node and execution client is expected.
2. gRPC is language agnostic.
3. Execution clients can be scaled more easily over multiple machines.

Another notable aspect is that the node and execution client need to both be client and server interchangeably. This is apparent in the following cases:

- Hornet as a Client: When Hornet detects a new transaction, it forwards the data over to iota-execution for processing.
- iota-execution as a Client: When requiring additional ledger data, iota-execution sends the request(s) to Hornet.
- Hornet as a Server: Responding to iota-execution requests, Hornet provides the required ledger data.
- iota-execution as a Server: After processing, iota-execution sends the results back to Hornet.

Thus, a gRPC bi-directional channel is defined as such in Listing 3.9:

Listing 3.9: gRPC bi-directional channel service

```
// The bi-directional stream for communication between Hornet and iota-execution.

service IotaExecutionStreamingGrpc {
    // A bi-directional streaming RPC initiated by Hornet and implemented in iota-execution.
    rpc ExecutionStream(stream NodeMessage) returns (stream ExecutorMessage);
}
```

iota-execution

The iota-execution component is responsible for the execution of Move transactions and is able to do so by receiving transactions on-demand via the previously mentioned gRPC interface.

The execution engine takes the following arguments as input:

- The transaction to execute,
- The corresponding milestone,
- Reference to the ledger view on which the transaction is to be executed.

The result of an execution results in either a legal or illegal transaction:

- **Legal transaction:** The execution engine returns all required changes to Hornet which commits it to the ledger including serialization of touched objects.
- **Illegal transaction:** The execution engine returns an error code to Hornet. Example of such transaction could be a conflict such as trying to mutate an already mutated object, or mutating an already deleted object.

See the Transaction Execution Flow in Appendix C to get a better understanding of the execution process.

inx-move

Hornet INX is a service that functions as a bridge between the Hornet node and external applications by using gRPC. To get INX ready for the MoveVM prototype, the existing INX is extended minimally, just enough to support the execution of Move transactions. This modified/extended version is called inx-move.

These added functionalities are:

- SubmitTransaction: Allows clients to submit Move transactions to the node for execution.

- DryRunTransaction: The DryRunTransaction service allows clients to simulate transaction execution without actually applying the transaction to the ledger. Useful for testing and debugging purposes.
- Reading and listening to Objects.

The Protobufs The protobuf definitions are hosted in the inx-move repository. They are used for Hornet INX, Hornet node and the iota-execution engine. The repo contains protobuf definitions for the following elements:

- **blocks**: Reading blocks and metadata from the node.
- **conflicts**: Definition of errors that lead to a conflict/illegal transaction.
- **effects**: Definitions of the result of a transaction execution for external tools. Hornet stores the serialized version of this for each legal transaction.
- **errors**: Move language errors that result in aborted transactions. Used only for external tools.
- **events**: Move events that are emitted during transaction execution. Used only for external tools.
- **ids**: Definition of transaction and object identifiers and references.
- **ledger**: Definitions of the ledger records and updates. These represent the types used directly in Hornet's ledger and database layer.
- **milestones**: Definitions of milestones and milestone metadata, furthermore WhiteFlag requests triggered by the coordinator.
- **move_lang**: Helpers for the move language.
- **objects**: Move level object definitions plus wrapped types for serialized objects used during WhiteFlag.
- **storage**: Intermediate types for the storage layer.
- **transactions**: Transaction types and helpers. Hornet only understands `RawTransaction` that is a serialized version of the transaction.
- **iota_execution**: Defines the bidirectional gRPC communication between Hornet and the execution engine. Contains messages for the state machine and the execution results.

iota-json-rpc
The JSON RPC provides a way of reading and writing data from and to the node. The server is connected to the indexer instance which uses mongodb. This indexer continuously syncs up with Object storage, Transactions and Events. It is being used by the IOTA CLI to interact with the node.

iota-cli
The IOTA Command Line Interface is a tool that facilitates interactions with the MoveVM and object ledger. It is designed to provide an easy interface to the JSON RPC service. Users can install this tool by building the rust binary and can then be used to for example send transactions.

iota-faucet
The IOTA Faucet runs as a separate service which can be used to distribute MANA tokens to users. Users can make use of this service by calling a command on the iota-cli. These tokens can be used to pay for transaction fees.

3.3.3. Transaction Execution and WhiteFlag

Hornet starts with the confirmation process once a milestone is reached. This process involves running the WhiteFlag algorithm, which traverses the corresponding milestone cone.

This milestone cones (green) are visualized in Figure 3.8. They consist of all the blocks between two milestones.

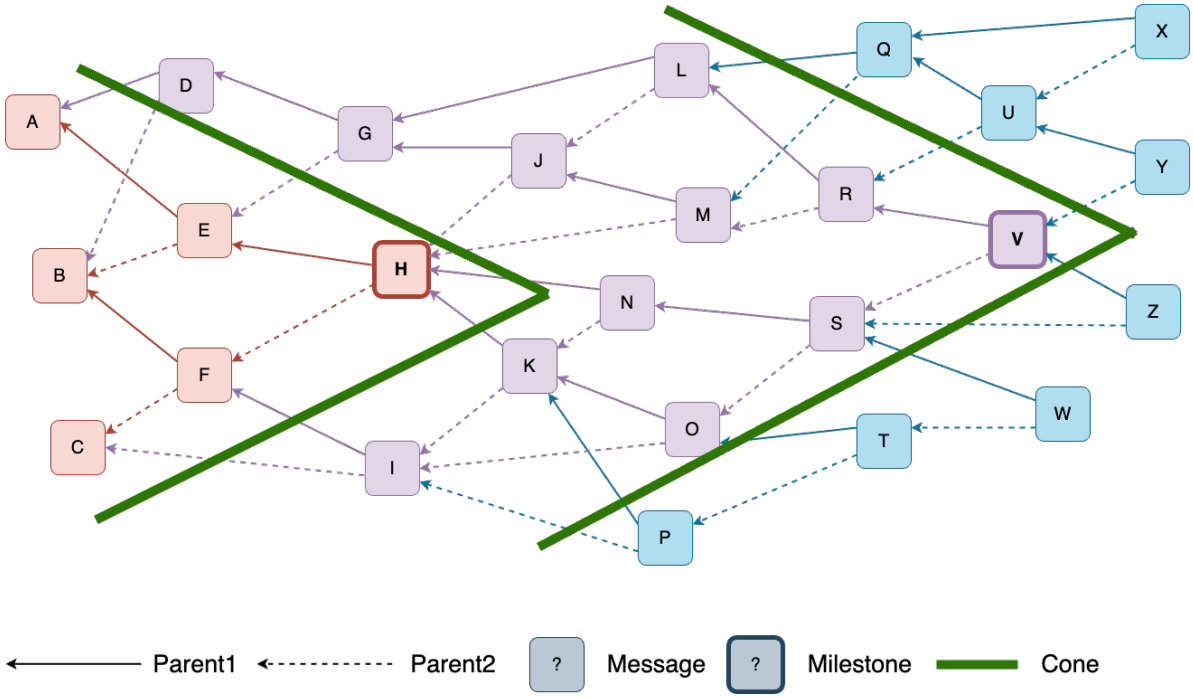


Figure 3.8: Milestone Cones, Source: IOTA Foundation (Modified)

When a transaction is encountered in the walk, an instance of `Executor` (execution client) is started. This `Executor` instance prepares the Hornet state machine and sets up the gRPC stream for communication with the execution environment. After initialization, Hornet forwards the specified transaction to the execution environment to be processed.

After iota-execution receives the transaction from Hornet, it validates it. This phase verifies the transaction's adherence to the required protocol, ensures it has a valid signature, and retrieves necessary objects. Should an error arise during the retrieval of objects, it is propagated back to the Hornet state machine, which may stop the process if the issue cannot be resolved.

The execution of the transaction begins in the execution environment after the validation phase. This execution phase involves the potential querying for additionally required objects. A transaction can be legal or illegal based on the execution result.

When an execution is successful, it is called a legal transaction result and contains all necessary changes that will be committed to the ledger.

The legal transaction result is composed in protobuf as such in Listing E.3:

Listing 3.10: LegalTransactionResult message

```
message LegalTransactionResult {
        types.id.TransactionId transaction_id = 1;
        bytes effects = 2;
        bytes events = 3;
        types.object.WrittenObjects written = 4;
        types.object.DeletedObjects deleted = 5;
    }
```

When a transaction is aborted, it is also seen as a legal transaction, as the used gas fees will still have to be written off.

An illegal transaction will have the following result, as seen in Listing 3.11:

Listing 3.11: IllegalTransactionResult message

```
message IllegalTransactionResult {
        types.conflict.ConflictError conflict_error = 1;
    }
```

This only contains a `ConflictError` which contains the conflict that occurred. Hornet marks it as a conflict using the WhiteFlag algorithm.

This dual-state machine process ensures that both Hornet and the execution environment work in tandem, from the initial transaction request to the final commitment to the ledger, ensuring a secure, consistent, and conflict-free ledger state.

3.3.4. Transaction ordering with WhiteFlag

The WhiteFlag algorithm is a transaction sequencing mechanism, used to maintain consistency between Hornet nodes that implement the Stardust protocol in IOTA [46]. The Hornet fork contains a modified WhiteFlag implementation to work with an object-centric ledger and interact with the MoveVM. The implementation was modified to:

- Recognize Move transactions.
- spin up a bi-directional gRPC stream to the execution environment.
- Request for execution.
- Correctly process resulting object mutations (written objects, deleted objects) and conflicts.

Transactions frequently need complex interactions with multiple objects. Special care was taken by testing that resulting object mutations and conflicts arising from the execution of Move transactions are correctly processed.

3.4. Implementation With Hornet

3.4.1. Introduction

This chapter contains the details of the implementation of all components described in the previous chapter. The inner workings of these components are discussed in great detail.

3.4.2. Hornet Node Software

The Hornet node software version v2.0.0-rc.6 has been forked and modified to support the execution of Move transactions. This new version is called hornet-move.

Modifications

There are a number of modifications made to the forked node:

1. Treasury, migrations and receipts have been removed.
2. The model of the ledger state has been modified by removing the UTXO logic.

   This has been done through a number of modifications:

   (a) The Output model has been replaced by the Object model.

   The side by side comparison in code between the old model and new model can be found in Appendix E Figure E.1.

   From this object model it is clear that the node only is aware of attributes like the ID and Sequence Number. The actual Move object data is included in the model, but it is serialized with BCS, which is handled by MoveVM.

   (b) The `Spent` model has been replaced by the `Deleted` model.

   Likewise, the side by side comparison in code between the old and new model can be found in Appendix E Figure E.2.

(c) The interface to model the UTXO storage operations, i.e., the `Manager`, has been replaced by an `Object` counterpart, e.g.:

Old Code

```
func (u *Manager)
    ReadOutputByOutputIDWithoutLocking
    (outputID iotago.OutputID) (*
    Output, error) { }
```

New Code

```
func (m *Manager)
    ReadObjectWithoutLocking(id
    iotago.ObjectID) (*Object, error
    ) { }
```

(d) The database prefixes `Output`, `Spent Output` and `Unspent Output` were replaced for their Object counterparts `Object`, `Deleted Object` and `Alive Object`.

The difference in Database structure can be found in Appendix E Figure E.3.

3. The transaction format in iota.go has been replaced by the Move transaction format. All UTXO related models, transactions, types and addresses have been removed as well.

   Likewise, the old and new code can be found in Appendix E Figure E.4.

   Notably, the full validation of a transaction does not happen in the node, but rather by the Execution layer. A transaction is only validated syntactically by the node by checking if all fields of the transaction model are present. The Execution layer does the semantic validation of the transaction by for example checking if an object that is given as input actually exists. This is due to the Move object being serialized in BCS format, which the Execution Layer is deserializing.

   A `ConflictError` type has been added to iota.go-move, so that rich expressions of conflicts can exist. The `ConflictKind` and `ConflictError` types can be (partially) found in Appendix E Listing E.4.

4. All tests have been removed related to UTXO manipulation. These have been replaced by tests involving Objects and transactions.

Whiteflag confirmation algorithm

The WhiteFlag confirmation algorithm also had to be changed. At the block level, everything remains the same. It traverses a milestone cone, collects all unreferenced blocks, and after validation of these blocks, it marks them as referenced. What changed about the algorithm is the way the ledger is being called and updated through transactions. To call MoveVM for the validation and execution of transactions, it is necessary to have a bidirectional communication channel to the Execution Layer.

Bidirectional communication channel A communication channel was established to facilitate interaction between Hornet-Move (implemented in Go) and the Execution Layer (implemented in Rust).

- When Hornet-Move needs to process a new transaction, such as during the execution of the WhiteFlag confirmation algorithm, it operates as a client, sending relevant data to the Execution Layer for processing.
- Upon processing completion, the Execution Layer assumes the server role, sending the results back to Hornet-Move.
- Conversely, when the Execution Layer requires additional ledger data, such as the state of a specific object at its latest version, it initiates requests to Hornet-Move, functioning as the client.
- In response to these requests from the Execution Layer, Hornet serves as the server, providing the necessary ledger data.

This channel is realized with gRPC where at both sides a state machine is in place. Figure 3.9 shows the overview of these state machines. Both are described in further detail as follows:

- **The hornet-move state machine:**

1. **Init State** - The first state where hornet-move creates the server-side state machine and requests the creation of a bidirectional channel to the Execution Layer. It transitions into the `ExecStart` state.
2. **ExecStart State** - The state entered into when hornet-move sends the transaction bytes to the Execution Layer. It can transition into the DBQuery state or the `ExecFinish` state.
3. **DBQuery State** - The state entered into when the Execution Layer requests objects from the ledger state. It processes the object request, fetches the object from the ledger db, and returns it to the Execution Layer. When a response is sent, it transitions into the previous state.
4. **ExecFinish State** - The state entered into when the Execution Layer sends the validation/execution results (illegal/legal with errors/legal successful). It marks the end of the hornet-move state machine.

- **The Execution Layer state machine:**

1. **StandBy State** - The first state where the Execution Layer waits for the requests of the creation of a bidirectional channel from hornet-move. It transitions into the `ValidateStart` state.
2. **ValidateStart State** - The state entered into when hornet-move sends the transaction bytes to the Execution Layer. It syntactically and semantically validates the transaction, i.e., it checks if it is legal. It can transition into the `ObjQuery` state or the `ExecStart` state or the `EndValidationError` state.
3. **ObjQuery State** - The state entered into when the Execution Layer requests objects from the ledger state and waits for a response. When a response is received, it transitions into the previous state.
4. **ExecStart Stat**e - The state entered into when the validation finishes and the transaction is legal. It executes the transaction and fetches objects from the ledger state if necessary. It can transition into the `ObjQuery` state or the `EndSuccess` state or the `EndAborted` state.
5. **EndValidationError State** - The state entered into when the validation provides an illegal result and sends it back to hornet-move. It marks the end of the Execution Layer state machine.
6. **EndSuccess State** - The state entered into when the execution provides a successful result and sends it back to hornet-move. It marks the end of the Execution Layer state machine.
7. **EndAborted State** - The state entered into when the the execution provides an error result and sends it back to hornet-move. It marks the end of the Execution Layer state machine.

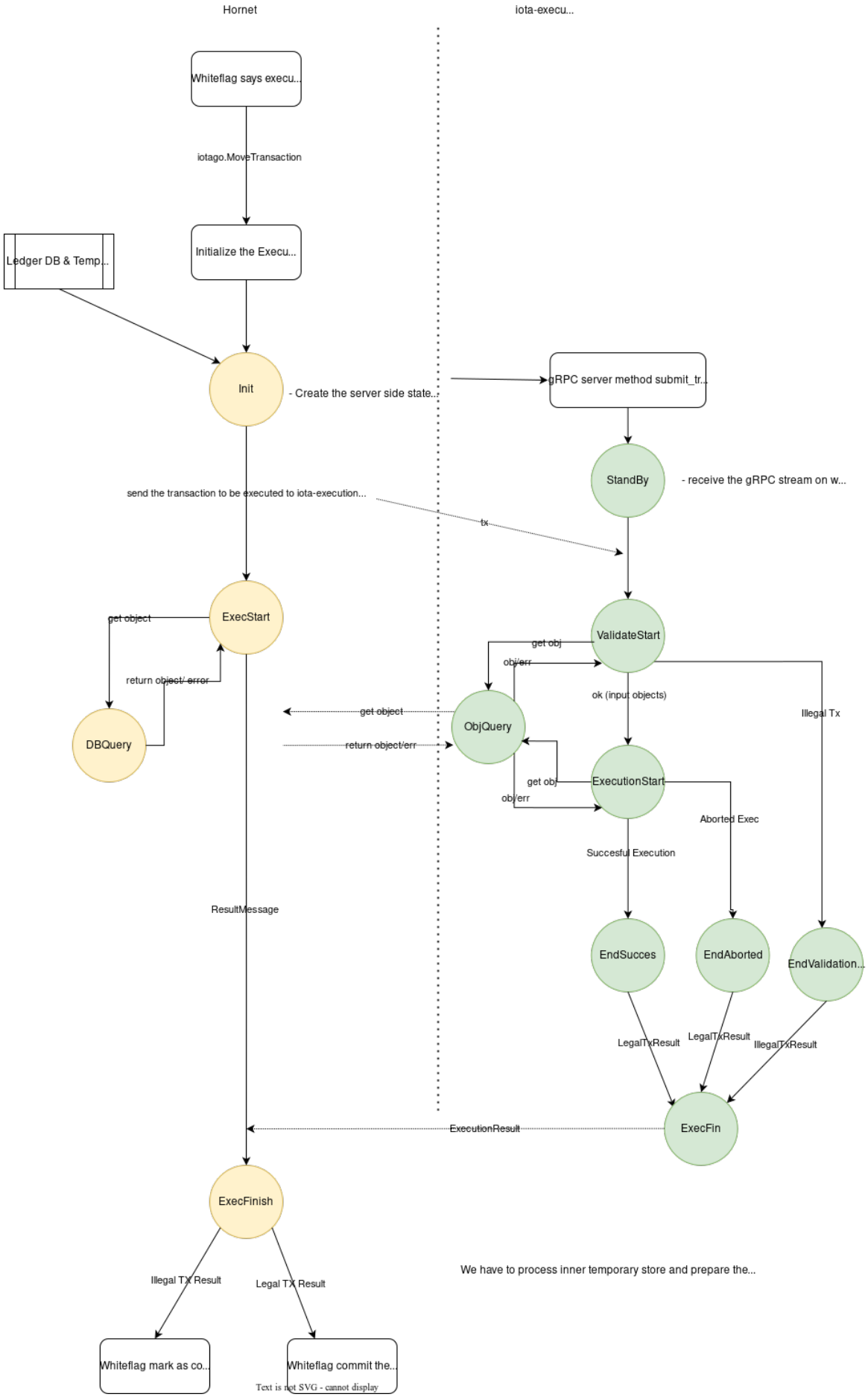


Figure 3.9: Statemachines: Hornet-move and Execution Layer

Validation and execution of transactions First, the confirmation algorithm generates a temporary store for object mutations and the blocks they reference. This temporary store is initialized as `wfConf`, as shown in Listing 3.12.

Listing 3.12: Temporary Store for Object Mutations in WhiteFlag algorithm

```
wfConf := &WhiteFlagMutations{
  ReferencedBlocks: make(ReferencedBlocks, 0),
  WrittenObjects:           make(map[iotago.VersionedObjectID]*object.Object),
  DeletedObjects:           make(map[iotago.VersionedObjectID]*object.Deleted),
  writtenObjectIDToVersion: make(map[iotago.ObjectID]iotago.SequenceNumber),
  deletedObjectIDToVersion: make(map[iotago.ObjectID]iotago.SequenceNumber),
}
```

Next, a post-order depth-first search is employed to traverse the approved blocks within the designated milestone cone. In this traversal, the blocks are applied to the temporary repository (representing the previous ledger state) in the sequence corresponding to their order. For each block:

- If the block does not contain a transaction, just mark it as referenced;
- Else, enter into the `Init` State of the state machine;
- Send the transaction bytes included in the block to the Execution Layer through the bidirectional channel;
- Receive the processing result exiting from the `ExecFinish` State of the state machine:
  - If the transaction is Illegal, mark the block as a conflict;
  - Else if the transaction is legal:
    - Get the transaction id, transaction effects BCS bytes and transaction events BCS bytes and store them into the block metadata in the temporary store (`WhiteFlagMutations`);
    - Get the written objects, i.e., created, modified and unwrapped objects, and store them in the temporary store (for legal NOT successful results, only an object will be present in this list, it is the gas object where the failing execution gas cost has been deducted);
    - Get the deleted objects, i.e., deleted and wrapped objects, and store them in the temporary store;

Then, the `InclusionMerkleRoot` and `AppliedMerkleRoot` is calculated.

Lastly, the algorithm updates both the ledger state and the metadata content of blocks in the node's storage by incorporating the mutations from the temporary store (`WhiteFlagMutations`). This process ultimately invokes the object storage manager's `ApplyConfirmationWithoutLocking` implementation, which can be found in Appendix E Listing E.2.

Transaction execution results There are two types of execution results:

1. **Illegal transaction:**
   When the transaction validation fails, the transaction is considered illegal. The Execution Layer returns to the WhiteFlag confirmation the failure in the form of a `ConflictError`, which is then saved into the transaction's block metadata. This scenario does not alter the ledger state. A `ConflictError` can include one of the following errors:

   - `InvalidSignature`
   - `TransactionDeserializationError`
   - `TransactionVersionNotSupported`
   - `SizeLimitExceeded`
   - `ObjectNotFound`
   - `DependentPackageNotFound`
   - `ObjectDeleted`
   - `MutableObjectUsedMoreThanOnce`
   - `MovePackageAsObject`
   - `MoveObjectAsPackage`
   - `InvalidObjectDigest`
   - `ObjectVersionUnavailableForConsumption`
   - `InvalidChildObject`
   - `IncorrectUserSignature`
   - `ObjectSequenceNumberTooHigh`
   - `GasObjectNotOwnedObject`

- `GasBudgetTooHigh`
- `GasBudgetTooLow`
- `GasBalanceTooLow`
- `InvalidGasObject`
- `InternalIotaExecutionError`

2. **Legal transaction:**
   If a transaction passes the validation, it is a legal transaction. This legal transaction can have two types of execution results:
   - **Successful Execution:**
     This means that MoveVM executed the transaction successfully. This case does not the ledger state. The Execution Layer returns to the WhiteFlag confirmation the results in the form of a message `LegalTransactionResult`, which can be found in Appendix E Listing E.3.
   - **Aborted Execution:**
     Means that MoveVM has thrown an error and aborted the execution. This case does not alter the ledger state. It works the same as the `Successful` Execution case, but the only modified object is the Gas Object given as input (deducting the gas budget).

INX
The IOTA Node Extension (INX) is a gRPC interface that allows for other applications to directly communicate with the node [47]. The original IOTA INX has been forked (inx v1.0.0-rc.2) and modified into the inx-move version. This version now makes it possible for Move and for Move related information to be retrieved from the ledger.

The INX Service Methods concerning node operations, milestone retrieval, block submission/reading tips, and REST API have not changed. All UTXO related was also replaced by Object models:

Blocks:

The information in blocks has been changed. The changes between old and new code can be found in Appendix E Figure E.5.

In this new implementation, the following new rpc methods are created:

- **ReadObject**: Enables to read from the node storage the latest version of an object indexed by the passed `ObjectID`. The implementation of this method ultimately resorts to the object storage's `ReadObjectWithoutLocking` and then checks and returns whether this object is Alive or Deleted.

  The shortened implementation can be found in Appendix E Listing E.5.
- **ReadObjects**: Returns the latest version of all objects that are currently stored in the node and that are not deleted. This is conceptually similar to `ReadUnspentOutputs`, i.e., the part of the ledger that can be used, but it assumes a new meaning in the case of objects. The implementation of this method ultimately resorts to the object storage's `ForEachAliveObject`, i.e., it gets the last version of all the `AliveObjects` that are the ones that have not been deleted yet;
- **ListenToLedgerUpdates**: Returns created, modified, wrapped, unwrapped or deleted objects as a result of the WhiteFlag confirmation (i.e., when the ledger is updated). The implementation of this method ultimately resorts to the object storage's `MilestoneDiffWithoutLocking` in order to return a `MilestoneDiff`, i.e., the summary of the changes that shall be applied to the ledger when a milestone is confirmed.

  The `MilestoneDiff` type implementation can be found in Appendix E Listing E.6

Transactions
Additionally, methods specifically related to transactions and execution have been added. All of these new rpc methods and corresponding messages can be found in Appendix E Listing E.7.

In this new implementation, the following new methods are created:

- **ReadTransaction**: Enables to read from the node storage the results, i.e., transaction effects and events bytes, of a legal transaction executed in the past and indexed by the passed `TransactionID`. The implementation of this method ultimately resorts to the transaction storage's `BlockIDByTransactionID` (described later in this section of the report) to be able to get the block (`CachedBlockMetadataOrNil`) that contains the `TransactionResults`. Listing 3.14 shows how this is implemented.

Listing 3.13: ReadTransaction implementation

```
blkID, err := deps.Storage.BlockIDByTransactionID(txID)
...
cachedBlockMetadata := deps.Storage.CachedBlockMetadataOrNil(*blkID)
...
t, err := NewTransactionWithResults(ctx, *blkID, cachedBlockMetadata.Metadata
    ())

```

- **ReadEvents, ReadEffects**: Enables to read from the node storage the events bytes and transaction effects as transaction results.
- **SubmitTransaction**: Submits a raw transaction into the node, that then takes care of creating the block, i.e., selecting block's parents and the protocol version and doing the Proof of Work. The block will be eventually referenced in a WhiteFlag confirmation. Listing 3.14 shows how this is implemented.

Listing 3.14: SubmitTransaction implementation

```
blockPayload := &iotago.MoveTransaction{BCSSerializedSenderSignedTx: make([]
    byte, len(rawTransaction.Data))}
copy(blockPayload.BCSSerializedSenderSignedTx, rawTransaction.Data[:])
block, err := builder.NewBlockBuilder().ProtocolVersion(protoParams.Version).
    Payload(blockPayload).Build()
...
blockID, err := attacher.AttachBlock(mergedCtx, block)

```

- **DryRunTransaction**: Skips the wait for a WhiteFlag confirmation and does not update the ledger. It just executes the transaction using the hornet-move state machine and the bidirectional communication with the Execution Layer and returning the results to the caller.
- **ListenToTransactions, ListenEvents, ListenEffects**: Work the same as their Read counterparts but continuously return all executed transaction results. Listing 3.15 shows how this is implemented.

Listing 3.15: Listen* Implementation

```
unhook := deps.Tangle.Events.BlockReferenced.Hook(func(blockMeta *storage.
    CachedMetadata, index iotago.MilestoneIndex, confTime uint32) {
  defer blockMeta.Release(true) // meta -1
  // referenced block doesn't contain a transaction, skip it
  if blockMeta.Metadata().IsNoTransaction() {
    return
  }
  payload, err := NewTransactionWithResults(ctx, blockMeta.Metadata().BlockID
    (), blockMeta.Metadata())
...
}

```

gRPC Types

Finally, when it comes to the types and models defined for hornet-move, inx-move can better understand Execution Layer types. This is due to inx-move defining gRPC types that directly relate to the

core types used by the Execution Layer. For example, while Hornet-Move is limited to working with `Object`, `TransactionEffects`, and `TransactionEvent` using their BCS representation, inx-move has a direct 1-to-1 representation as a gRPC type.

INX Plugins A few existing INX plugins have been modified to support the new Object model.

- **inx-app**: This is the implementation of an inx client, specifically created to provide with a predefined nodebridge interface and a httpserver. The ìnx-app v1.0.0-rc.3 was forked to create the inx-app-move version, that supports the inx-move interface.
- **inx-coordinator**: This is a plugin that implements the function of a Tangle Coordinator node. The inx-coordinator v1.0.0 was forked to create the inx-coordinator-move version, that supports the inx-move interface. No functional modifications were made, only types and dependencies were updated to support the move flavor.
- **inx-spammer**: This is a plugin that implements the process of spamming blocks into a Tangle. The inx-spammer v1.0.0 was forked to create the inx-spammer-move version, that supports the inx-move interface. The plugin was modified to spam only tagged data block payloads. It means that it is not able to spam blocks containing transactions.
- **iota-faucet, iota-json-rpc (indexer)**: These are two inx-move plugins, created to specifically operate with Move objects. Their specific functioning is discussed later in this chapter.

The Object Ledger

Storage model additions The storage model has been modified by removing the UTXO-related logic. The rest remained unchanged apart from two models:

1. **BlockMetadata**: Has been expanded with transaction related information. This expansion is shown in Listing 3.16.

Listing 3.16: BlockMetadata type

```
type BlockMetadata struct {
  ...

  conflict iotago.ConflictError

  ...

  transactionID *iotago.MoveTransactionID

\\ TransactionEffects and TransactionEvents encoded in BCS
  transactionResult []byte
}

```

2. **Transaction**: This storage model has been added to bind a transaction id to a block id, i.e., a reverse index that was previously feasibly obtained through a block's outputs ids, but that is now unfeasible to obtain from object ids (the Object model as no reference to the block containing the transaction that manipulated it, but only to the transaction). This Transaction storage model is shown in Listing 3.17.

Listing 3.17: Transaction Storage type

```
type Transaction struct {
  objectstorage.StorableObjectFlags
  transactionID      iotago.MoveTransactionID
  transactionBlockID iotago.BlockID
}

```

Applying the transaction execution result to the storage During the execution of the WhiteFlag confirmation, a temporary object manager maintains written (i.e., created, modified, unwrapped) objects and deleted (i.e., deleted, wrapped) object states. The temporary object manager is a wrapper around the object manager plus the intermediate ledger state during WhiteFlag, as can be seen in the implementation 3.18:

Listing 3.18: TemporaryObjectManager type

```
type TemporaryObjectManager struct {
  LedgerStateManager *object.Manager
  Mutations          *WhiteFlagMutations
}
```

Each request to fetch the latest state of an object during the WhiteFlag confirmation transaction execution passes through this component.

Listing 3.19 shows the implementation of the `ReadObject` method.

Listing 3.19: ReadObject method

```
func (t *TemporaryObjectManager) ReadObject(id iotago.ObjectID) (*object.Object,
    error) { ... }
```

The `ReadObject` method reads the most recent version of an object from the temporary ledger state or, if this has not been touched by any transaction during the current WhiteFlag confirmation, from the object manager.

This mechanism allows for a complete rollback of the transaction effects during the WhiteFlag confirmation in case of error.

After a successful execution of the WhiteFlag confirmation, the new versions of the written and deleted objects are pushed to the storage. The implementation for this can be found in Appendix E Listing E.8. Then, the metadata of all the transactions contained in the WhiteFlag referenced blocks are updated accordingly, i.e., either with a conflict error, if they are illegal, or with a transaction result, if they are legal. The implementation for this part of the code can be found in Appendix E Listing E.9.

Pruning Two functions are responsible for pruning data:

1. **pruneMilestone**: Removes the `MilestoneDiff` for the given milestone index and all objects that were deleted in this milestone; it ultimately resorts to the object storage manager's method `PruneMilestoneIndexWithoutLocking`. The implementation of this function can be found in Appendix E Listing E.10.
2. **pruneBlocks**: Removes all the associated data of the given block IDs from the database plus the association between transaction id and block id formed in the Transaction storage. Likewise, the implementation can be found in Listing E.11.

Snapshots A full snapshot contains the ledger objects as of the Confirmed Milestone Index (CMI) and the milestone diffs from the CMI back to the snapshot's target index.

The ledger objects are all the so-called ”alive” objects, i.e., all the created objects that have not yet been deleted.

Genesis state generation

The genesis is the starting point of a new network, block zero. In our new model, it is the process of creating all the objects necessary to bootstrap the Tangle. This includes the packages for iota-framework, which are fundamental Move modules, plus some initial objects for economic transactions in the Tangle.

The genesis process is predefined in the code. Specifically in the iota-genesis library, which invokes a function in the iota-framework called `iota::genesis::create_genesis`. This function does the following:

1. The object including all the iota-framework modules is created.
2. The IOTA tokens are minted. This creates a new object containing an IOTA coin balance.
3. The MANA tokens are minted. This creates a new object containing a MANA coin balance.
4. All assets are transferred to a preset address that represents the faucet (described later in this chapter).

3.4.3. Execution Client

The execution layer handles the execution of Move transactions, which happens on a separate layer. This layer is responsible for validation on various levels, and for the evaluation of the transaction effects. It is the node that applies or not those effects on the ledger state. Hence every transaction execution can be construed as a dry run. This layer is referred to as iota-execution: it consists of a Rust library that uses iota-adapter, and on top of that, a Grpc server able to handle transactions encoded in raw bytes is built.

The execution layer might be detached from the node, but is not independent. Executing a transaction requires querying the ledger state for objects and packages, and this is attained through the bidirectional communication established between iota-execution and the hornet node. This bidirectional communication has a limited lifetime spanning the time that the transaction is sent to iota-execution, until it is validated and its effects are evaluated to be finally sent back.

Binary canonical serialization (BCS)

Hornet is agnostic of the representation of objects and transactions in the execution layer, and the latter is agnostic of their representation in the Move VM.

To accommodate the transfer of values between the three components, the binary canonical serialization (BCS) format is used, that is extensively used in most of the blockchains based on Move (Sui, Aptos). BCS provides concise binary representations guaranteeing that for every value of a given type there is only one valid representation. The latter property, in particular, has benefits in applications to cryptography, and this reflects on how transaction signatures are evaluated.

More specifically, BCS is used in the following cases:

- Transactions and objects are represented as BCS bytes in Hornet.
- Transaction signatures are evaluated on the underlying BCS representation
- Arguments passed to smart-contract entrypoints are serialized using BCS. This includes primitive values, as well as objects of any kind.

Move transaction kinds

Support is available for two different kinds of transactions:

1. **Move-call transactions**: These encapsulate all the data necessary to invoke entrypoints of published move package modules. This includes an identifier of the entrypoint, the specification of any generic type arguments if applicable and arguments to the entrypoint function.
2. **Publish transactions**: these encapsulate all the data necessary to publish new move packages, including the compiled package modules serialized according to the move binary file format.

Each transaction is always submitted and signed by an address, which is referred to as the signer. This address also always adds a gas object to pay for the fees. A transaction is accompanied by additional input objects and transaction context which is referred to as `TxContext`.

Input objects are all objects pertaining to the scope of the transaction: Gas objects, dependent packages for publish transactions, or the calling package and its dependencies for move-call transactions, plus any object that is part of the calling argument for an entrypoint function. These objects are resolved by making queries to the node in order to fetch them from the ledger state.

The `TxContext` is a special value that can be thought of as a factory of object IDs during the execution of the transaction. It hashes the signer address, the transaction digest, and the milestone data (index and timestamp), along with an incremental index to derive any new object ID required.

Transaction processing: Validation

A transaction goes through a validation process after it is received from the node. Validation starts with the deserialization of the encoded transaction, and thus it is ensured that it is syntactically correct. Then it is verified that the transaction signature is valid, which guarantees that the sender of the transaction has indeed signed the transaction data.

Subsequent operations guard against errors on common and kind-specific transaction input like missing gas payment, or arguments being in conflict with size limits imposed by the execution protocol. Examples include exceeding maximum number or depth of type arguments, maximum number of function arguments or modules to publish etc.

Finally, additional checks are made while resolving the input objects from the ledger state. These checks extend in depth and breadth, and include object-kind checks, ownership checks, object integrity checks etc. The most prominent failure in this stage of the validation is when the version of an object declared as input is not found, or is already outdated in the ledger.

Validation failures are classified as conflicts, and the respective transactions are classified as illegal, which causes the transaction to not be executed.

Transaction processing: Execution

Transactions that pass validation are considered legal and forwarded for execution. Execution involves using the adapter, that acts as a high-level interface to the Move VM, to eventually invoke the bytecode instructions that correspond to the transaction under process. In the process the necessary gas is charged for computation and storage costs, and the transaction effects are evaluated.

Transaction effects and events It is already discussed that execution on this layer does not affect the ledger state. Instead, an isolated temporary view on the ledger state is created before initiating the execution. This view is referred to as the temporary store, and initially contains the input objects of the transaction.

As the transaction is executed in the VM, the temporary store is updated to record the projected changes in the ledger state. Two types of changes are tracked: writes and deletions.

Writes include mutations of input objects (e.g. the gas object that pays for transaction costs), creation of new objects, and unwrapping of previously wrapped objects.

It should be noted, that objects that are simply read during the transaction are also classified as mutated, and their version is bumped. There is an exception to this, depending of the kind of the object stored.

More specifically, objects in the ledger state belong to either of two kinds:

1. Move objects: struct values created from their defining packages
2. Move packages: programs that contain logic and struct definitions

Move packages are immutable so their version is not bumped.

Deletions include wrapping an input object, or removing it in the sense that it will not be present in the ledger state.

When execution finishes successfully, all changes are reported back to the node in the form of a set of written objects, and a set of deleted objects, with respect to writes and deletions recorded.

Due to the storage method of objects in the node, deleted objects include the outdated versions of mutated objects in the written set.

In addition to projected changes in the ledger state, any events emitted while executing transactions are also reported back to the node.

Gas metering Every operation that comes in effect during execution is associated to a gas cost. This is measured in abstract gas units that are associated to a quantity of MANA, according to the gas price defined on the adapter level. Operations that affect the size of the storage incur a storage cost, whereas computational operations incur a computation cost.

The cost is metered in various levels of the process. First gas is charged for reading objects from storage in order to create the temporary store. VM computations charge gas on the basis of a cost table that maps bytecode instructions to the associated cost. The cost table, as well as the exact metering scheme are defined on the adapter level, which provides the flexibility to define metering rules and invariant checks on top of the Move VM.

The computation cost up to this point is classified into buckets, i.e. bands of gas units that have a fixed cost. E.g. costs that fall to the 1001 - 5000 range assume a universal cost of 5000 units, so that the step function is not linear as the cost increases. This is to prevent obsessive gas optimization needs by developers of smart contracts.

Besides the VM operations, gas is charged for operations that are the result of interaction with the storage. Verifying and linking a package, reading or writing to the storage are associated to computational costs, while maintaining the contents of an object into storage brings about a storage cost.

Storage costs however are refundable. Once an object is deleted from the storage the cost expended for storing it the ledger state is refunded to the owner. This is referred to as the storage rebate.

Failures Execution might of course fail for various reasons. For example, when the gas payment provided by the sender of the transaction is not sufficient to cover the transaction costs.

In such cases, the sender is charged for the object reads required to populate the temporary store plus any accumulated computational cost up to the point of failure. Execution then terminates with no other effect than the mutation of the gas objects referenced by the transaction.

Dry running a transaction

It has been already pointed out that operations on the execution layer have no effect on the ledger state maintained on the node. Hence, every execution is essentially a dry run, evaluating the transaction effects.

The node gets back the transaction effects, and has the sole responsibility of applying them to the ledger state with WhiteFlag confirmation.

Thus when requesting the node to dry run a transaction, the node does not alter the ledger state, but only returns the transaction effects evaluated on the execution layer.

Adapter tests

Execution is tested by initializing a test adapter that instantiates a new VM, and running publish and call transactions for a set of simple test smart contracts to assert that the evaluated effects are the expected ones.

To this end, a testing framework has been developed: the iota-transactional-test-runner that relies on existing infastructure provided by the Move language.

Test cases are encapsulated in the iota-adapter-transactional-tests crate.

3.4.4. IOTA Move Framework

The IOTA Move Framework is a set of built-in modules that provide an on-chain API for Move developers. Some of these provide utility functions such as math and cryptography, while others provide basic functionality functions such as object management and transfers.

The framework can be found in the iota-framework crate. This framework contains the modules written in Move, but also some native functions written in Rust. It also contains a custom object runtime to facilitate objects.

The Move modules reside in two packages which have been published at genesis: std at 0x1 and iota at 0x2. These two packages are built and used by iota-genesis to create the genesis ledger state.

The native functions are not compiled and published on the ledger, rather, they are passed to the MoveVM upon its creation in iota-execution.

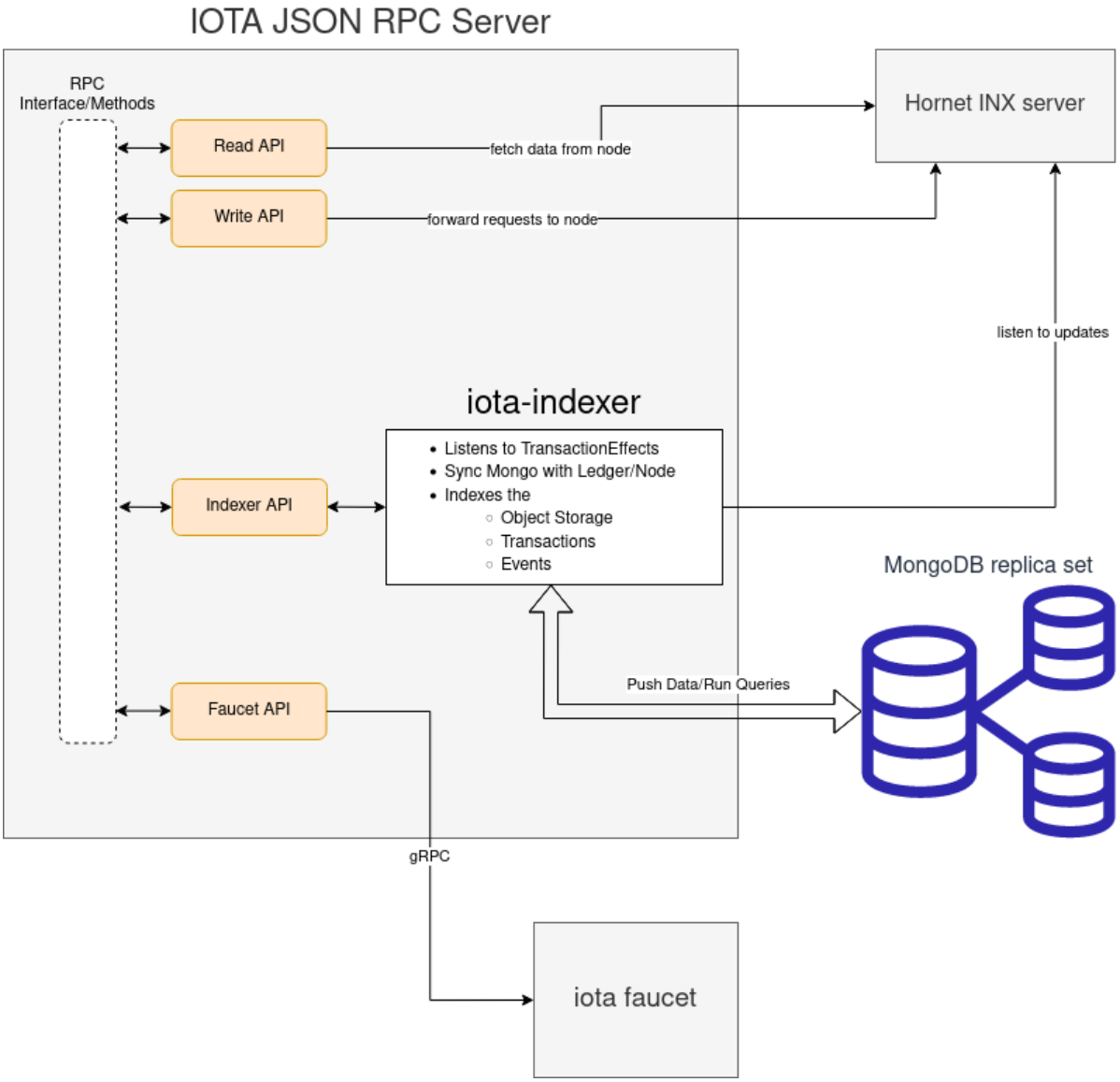


Figure 3.10: JSON-RPC

3.4.5. JSON-RPC Service

The data that is stored in Hornet is accessible through a JSON-RPC service. This service provides an API for retrieving and modifying that data. An overview of this service architecture is given in figure 3.10.

The supported methods can be found in this table:

Parallel to the JSON-RPC service, the indexer process stores transactions, effects, and events from the node into persistent storage. The indexer communicates with the node through INX and fetches all transactions that have not yet been indexed up to the latest milestone. This process is called synchronisation. It does so by listening for new milestones. Then, all new transactions are stored in the database, with the encapsulated data also being stored for easy access. Whenever the JSON-RPC service stops, the indexer also shuts down gracefully, preventing any missing data in the database.

The reason MongoDB is chosen for the database instead of PostGres, was due to the fact that MongoDB has a better suited model for storing data than PostGres. PostGres uses a relational model while MongoDB uses a documentation model. These models are visualized in figure 3.11.

It is easy to see that the relational model is a lot more complex than the documentation model. The

API Category	API Description
Read API	
iota_getObject	Returns the object information for the specified object ID.
iota_getTransaction	Returns the transaction information for the specified transaction digest.
iota_getEvents	Returns the events of a transaction for the specified transaction digest.
Write API	
iota_dryRunTransaction	Dry runs a transaction on the node and returns the effects without committing the underlying changes in the ledger.
iota_executeTransaction	Executes a transaction on the node and returns the effects after committing the underlying changes in the ledger.
Indexer API	
iotax_getOwnedObjects	Get objects from the indexer DB based on multiple filters.
iotax_getTransactions	Get transactions from the indexer DB based on multiple filters.
iotax_getEvents	Get events from the indexer DB based on multiple filters.
Faucet API	
iota_requestGas	Request gas for the specified wallet address.

Table 3.2: APIs and Descriptions

latter is also a lot easier to maintain and use for prototyping, because the data that is stored changes a lot during prototyping. Therefore, it is decided to implement MongoDB for the database needs. It is also decided to implement data replication so that it would be possible to run atomic transactions across multiple documents, i.e., doing multiple database interactions at the same time.

The indexer supports the following query filters:

1. Transactions
   (a) Matching the given transaction digests.
   (b) Sent by a wallet address.
   (c) Paid the gas object with the given object ID.
   (d) Of a particular kind.
   (e) Calling a particular package, module, or even function.
2. Objects
   (a) Matching the given object IDs.
   (b) Owned by a wallet address.
   (c) Of a particular kind (move object or package).
   (d) Of a particular struct type (e.g. `iota_framework::coin::Coin<Meta>`).
   (e) Of a particular version.
3. Events
   (a) Emitted in transactions sent by a given wallet address.
   (b) Defined in the given package, or module.
   (c) With the given name.

3.4.6. Faucet

The iota-faucet is an INX plugin which exposes a method for requesting a fixed amount of MANA coins. As all gas fees need to be paid with MANA coins, this component helps to acquire the coins needed for ledger interactions.

This faucet is instantiated during genesis as shown in Listing 3.20.

Listing 3.20: Faucet Initialization

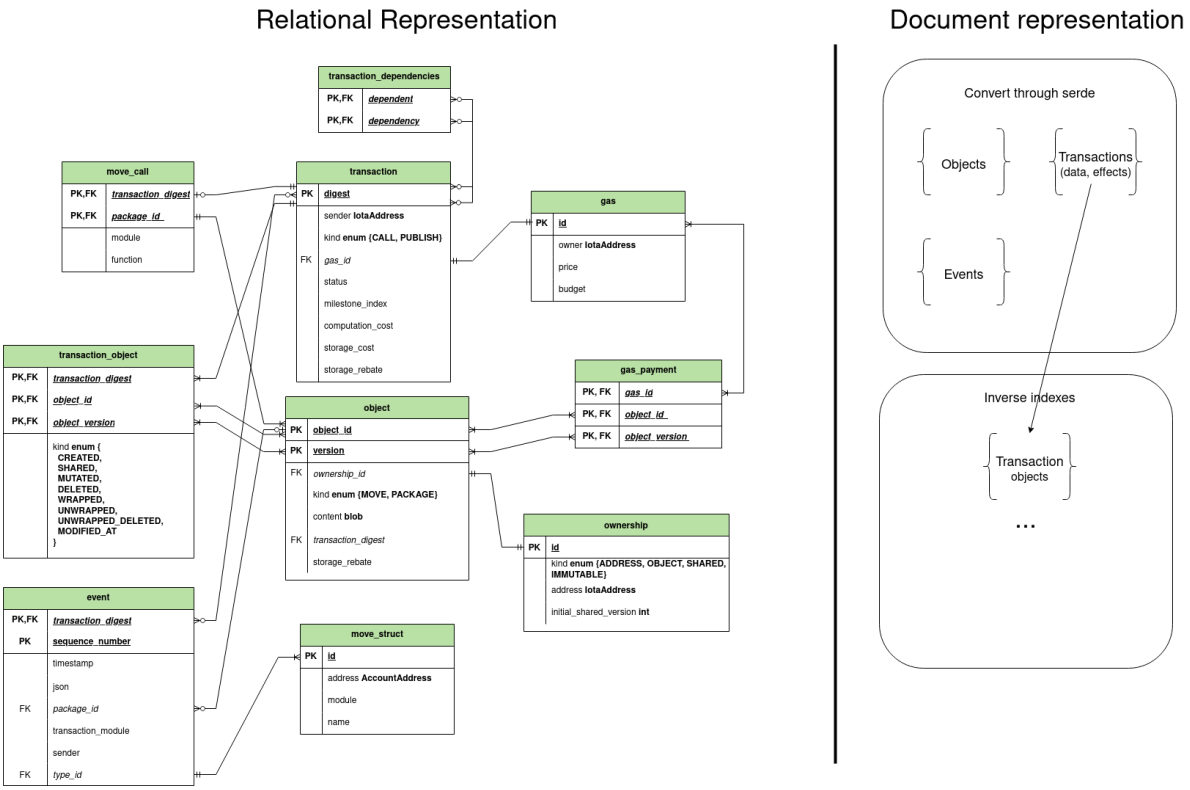


Figure 3.11: DB Models

```
// create a liquidity pool for exchanging IOTA and MANA coins
let lp_balance = mana_pool::init_pool(p, iota_liquidity, mana_liquidity,
    MANA_POOL_FEE_BASIS_POINTS);
// get the coins objects references
let iota_coin = from_balance<IOTA>(minted_iota, ctx);
let mana_coin = from_balance<MANA>(minted_mana, ctx);
// split MANA coins object in two, in order to have a new object that pays for the
     faucet execution gas
// -> mana_gas_coin
let mana_gas_coin = split<MANA>(
    &mut mana_coin,
    available_mana_tokens / MAX_BASIS_POINTS * GAS_RESERVE_RATIO,
    ctx,
);
let lp_coin = from_balance<MANA_POOL>(lp_balance, ctx);
// distribute the coins to genesis addresses
distribute_tokens_to_wallets<IOTA>(&iota_distribution, &mut iota_coin, ctx);
distribute_tokens_to_wallets<MANA>(&mana_distribution, &mut mana_coin, ctx);
// transfer the remaining coins to the faucet
public_transfer(iota_coin, faucet);
public_transfer(mana_coin, faucet);
public_transfer(mana_gas_coin, faucet);
public_transfer(lp_coin, faucet);
```

In this faucet initialization code, the process starts with creating a liquidity pool for exchanging IOTA coins and MANA coins. Then, the object that contains the entire supply of MANA coins is retrieved and split into two objects. This new object is needed to pay for the faucet gas fees. After this, a fixed amount of IOTA and MANA coins is distributed to the genesis addresses. And finally, all the remaining

coins are being sent back to the faucet address.

Faucet algorithm

- The faucet implements an INX client for dialoguing with the hornet-move node.
- It gets the latest version of the mana_gas_coin object through the INX's read_object operation.
- It gets the latest version of the MANA coin supply object through the INX's read_object operation.
- It creates a new transaction with a Move call to the split_and_transfer method of the pay module.
  - The inputs are:
    - MANA coin supply object.
    - A certain amount indicated as an integer.
    - The address of the faucet user, i.e., the receiver; this address is generated from a mnemonic that is fixed for development purposes but that can be set as a configuration parameter.
  - This method splits the coin object passed as input and transfers the newly created object to the address indicated as the receiver.
  - The mana_gas_coin object is used to pay for the execution gas.
  - It uses the INX's submit_transaction method to create a block including this transaction ('s bytes) and submitting it to the hornet-move node.
  - It waits for the execution result by listening to referenced blocks through the INX's method listen_to_referenced_blocks and waiting for the issued block to be referenced.
  - It finally returns the object reference (i.e., a tuple containing the object id, sequence number, and object digest) of the newly created object containing a MANA balance for the faucet user.

Faucet API

The faucet can be called through numerous ways:

1. **gRPC**:
   - The faucet exposes the server to the port 6057 by default.
   - The server implements a gRPC interace with just one method, shown in Listing 3.21:

Listing 3.21: IotaFaucetGrpc service

```
service IotaFaucetGrpc {
  rpc ManaFixedAmount(ManaFixedAmountParams) returns (types.id.ObjectRef)
    ;
}
message ManaFixedAmountParams {
  bytes address = 1;
}

```

   - This method requires an address (in bytes) that will receive the MANA coin object as input and returns the newly generated MANA coin object reference as output.
   - For each request, a fixed amount of 1 MANA will be transferred to the requestor's indicated address.
2. **JSON-RPC**:
   - The faucet operation is exposed by the JSON-RPC API (described above).
   - A JSON-RPC client can access the faucet_api's method request_gas:

```
let response = client.faucet_api().request_gas(address).await?;
```

- This method requires an address (of JSON-RPC type IotaAddress) that will receive the MANA coin object as input and returns the newly generated MANA coin object reference as output.
- The implementation of request_gas is a gRPC client that invokes the ManaFixedAmount method described above. Thus, also in this case, a fixed amount of 1 Million MANA will be transferred to the requestor's indicated address.

3. **CLI**:
   - The CLI can be used to request gas: "iota client request-gas".
   - The CLI makes use of the JSON-RPC API to send a MANA coin object to the CLI's active address.

4

Move application: Intents

4.1. Introduction

Now that smart contracts became possible on L1, novel modifications can be designed that were not possible before. One of these is support for native on-chain intent execution. This chapter will show the design and implementation for a prototype of an application-layer intent module and additionally provide the design for a more complete novel intent-based execution pipeline. The latter will not be implemented but will only be theoretically designed due to the complexity of building such complete system and time constraints of this thesis. For example, Anoma, a project focusing on building an intent-based architecture, has been in development for years [48].

4.2. Intents

An intent is an expression of what a user wants to achieve whenever they interact with a protocol. Intents describe a desired state transition, but do not specify how to carry it out. Intents are not exact transactions as they do not have an exact execution path, they have a solver in between ask (intention) and outcome. The solver constructs the transaction for you. Figure 4.1 shows the difference between the execution paths of a normal transaction and an intent transaction. Between the outcome and input (the want) resides a matchmaking stage.

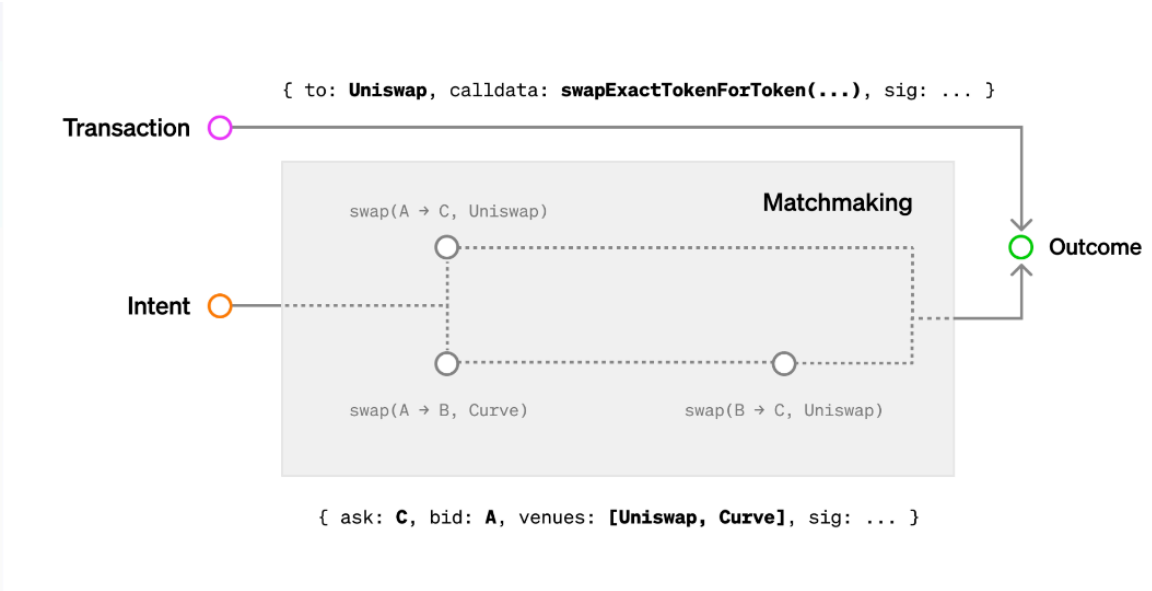


Figure 4.1: Transaction vs Intent execution path, Source: Paradigm

In an intent-based world, the solver does this matchmaking for you. Solvers are entities that construct transactions that fulfill user intent(s). For that it needs to know the constraints (intents), all available execution paths, and evaluate all alternative paths and choose which one is optimal in that specific situation.

4.3. Simple Intent-based architecture

The idea behind the simple intent-based architecture is as follows. We should be able to facilitate automatic execution of lined-up intents for a specific module. These intents are simply transactions that will be executed the most efficiently, or at least how the module dev wants it, which is usually in the best interest of the user. An example of such an intent functionality is a batch-swap for a dex. In batch swaps, the user wants their trade to be executed in the most efficient way possible, in terms of cost-efficiency.

Intents will just be objects filled with all necessary information, which is sent to the relevant queue. This is done so that intents are simply an addition to the current working transaction pipeline, and does not require any breaking changes. Once the intents entered the queue, they will emit an event and signal to the system that there are intents in the queue waiting to be processed. The system then sends out a system transaction calling the `process_queue()` function at the checkpoint, which happens each consensus round.

4.3.1. Design

There are numerous ways of realizing this into a basic prototype. The one chosen is a simple approach to this problem, and can be implemented with little modification to the core code.

After each milestone, the node traverses the blocks from previous milestone to latest new milestone. This is called the cone, which consists of transactions. The WhiteFlag algorithm traverses this cone to create a deterministic order for the transactions. Our hornet-move prototype executes all move transactions in the same order of the WhiteFlag traversal. Our simple approach to the intent queue processing is to check for events emitted by the relevant module, which are emitted upon enqueueing. If found, the system transaction is sent out to execute the intents from the queue.

4.3.2. Implementation

The module that will be used is the swap module in the `batch_swap` package. This module contains a basic pool that will be used to swap tokens from A to B, and from B to A. For simplicity sake, the intent-related functions are included in the same module.

For this example, we create a pool which allows for swapping between two tokens: MANA and USD. MANA we get through our faucet tool, and USD is our own token that we minted for this specific purpose. The minted USD is only deposited to our own address upon deployment of the USD module.

The complete code for the swap and usd modules can be found in Appendix B.

The user submits a normal transaction, which is an intent of its swap. This intent is then added to the queue. At the end of each milestone, the node checks for any intent-transaction in the sub-dag from last milestone to current new milestone. In the current implementation, this is simply checking for interactions with the add_intent function of that specific module. If there was an intent transaction, then the node submits a system-transaction which calls process_queue of that module. All intents in the queue will then be processed in a batch swap, in which each swap will only be done once. This `process_queue` function is written by the developer and can do with the intents as he wishes. In our example, it is a batch swap function, which combines all swap intents and calls the swap function once per swap direction. Figure 4.2 shows an overview of the sequence of an intent transaction.

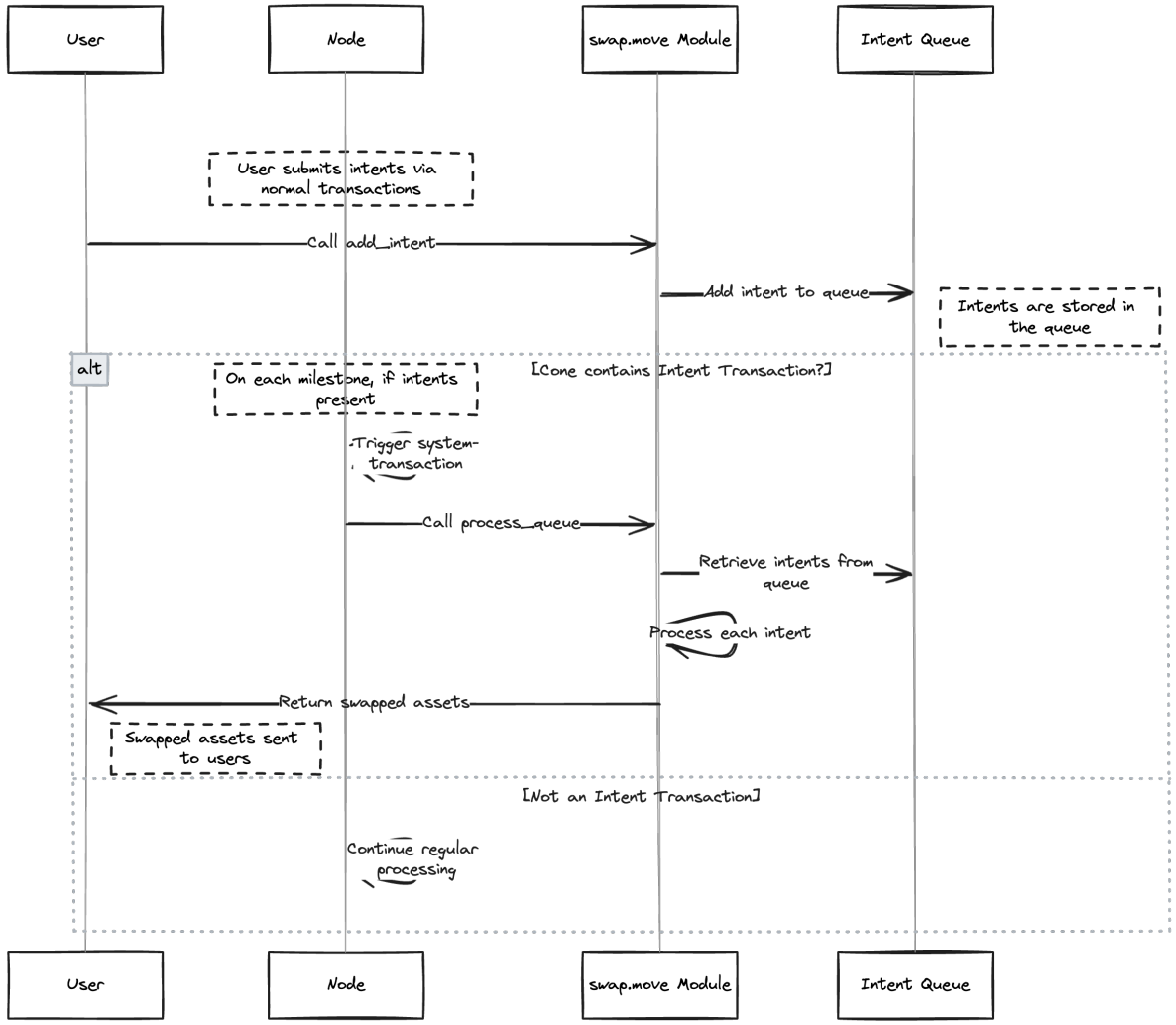


Figure 4.2: Sequence diagram of the batch swap intent processing.

4.4. Advanced Intent-based architecture

This chapter will introduce the design of a custom transaction execution mechanism for IOTA 2.0. We will use Sui's object-based move model and will design a custom execution pipeline in which the sequencing of transactions will be programmable through move modules. We will not implement this design due to time constraints.

4.4.1. Design considerations

The IOTA 2.0 protocol is not released as of date of writing. Thus, all design decisions will be made based on the currently available knowledge of the IOTA 2.0 design. During this time, changes to the IOTA 2.0 design are possible, which will affect the correctness of the adapter.

As this thesis will only focus on the adapter part of handling smart contracts on layer 1, other changes to, for example, the IOTA 2.0 consensus will not be discussed in much detail. However, we will propose some modifications to a part of the consensus.

4.4.2. Consensus and Execution

The consensus algorithm of IOTA 2.0 utilizes on-tangle voting along with a reality-based ledger to facilitate the confirmation of transactions. This algorithm works well for UTXO-style transactions where you choose a winner in conflicts, but it cannot sequence transactions touching the same shared state or object. Therefore, we must think of a new way of making this work. The proposal is to leverage

Move modules for on-chain sequencing.

This will be realized in two steps, which will be elaborated further in the coming sections:

1. Slicing
2. Sequencing

Slicing

This part of the process will fetch a new chunk of the tangle that will be sequenced in the following step. This is already done in a way with the current coordinator milestones, in which slices are created with blocks between each milestone. Committed leader blocks create the beginning and end of each slice. These slices are similar to the milestone cones shown in 3.8.

As mentioned earlier, the consensus is not part of the adapter and will thus not be mentioned in more detail. From here on, we will treat the slicing as a mechanism that is taken for granted. For this design, and the further implementation of the adapter, we assume that the blocks received from the slicing part is an arbitrary set of blocks. This set of blocks can then be used in the next step.

Sequencing

The sequencing step takes the set of blocks, or slice, from the previous slicing step as input. Each block is simply a transaction. The output of the sequencing step is the order in which the transactions will be executed. Another thing that sequencing has an impact on is MEV, which stands for Maximal Extractable Value. In MEV, the sequential position of transactions in a block is being ”used” for personal monetary gain. An example of MEV is front-running, in which a transaction with a large monetary value on a decentralized exchange is being front-run by another transaction. This is usually done automatically by bots. These bots detect such opportunities by scanning mempools for these kinds of large transactions, and when they find one, they send a similar transaction but with a higher gas price, which places their transaction in front of the targeted transaction, essentially front-running it. The ”victim”-transaction of the front-running transaction will now execute on a different shared state of the decentralized exchange, usually in disfavour of the victim. Just like this example, there are many more MEV strategies that are being used, and as you can tell, these are usually not beneficial to ordinary users of the DLT.

Thus, the way we do sequencing is important to protect fairness between users of the DLT. We came up with a few different sequencing algorithms that we could use:

1. Random order. This is the simplest of the bunch. Transactions will just be shuffled in a random order.
2. Order based on transaction hash. This approach orders all transaction in a slice based on their transaction hash, which could be in ascending or descending order.
3. Order based on MANA fee. This is a more commonly seen algorithm to determine the execution order. A higher MANA fee means an earlier position in the order. This is also the algorithm used by SUI.
4. Preserve the whiteflag-order that comes with the walk on the blocks.

With these sequencing algorithms, we are still not able to completely get rid of MEV. In fact, in some cases it might even be beneficial for users. It could for example be used in the favor of users. MEV could be used as a service instead of a hindrance. Instead of extracting additional value from transactions, it could extract additional value for the transactions. Therefore, we came up with a novel approach for this sequencing issue: allowing application developers to choose and/or write their own sequencing algorithm. However, this approach comes with a consequence, which is that these applications give up on composability with applications using a different sequencing algorithm.

In this custom sequencing approach, every application developer can opt for their own sequencing algorithm but can also just use the default sequencer, which we call the `Orchestrator`. And the best part about this approach is that these sequencing algorithms are written in Move, and are part of the on-chain system modules. This means that upgrading the sequencing algorithm is also as simple as upgrading a Move module.

For simplicity sake and due to time constraints, we will partially implement the custom sequencing approach. The implementation will only contain one default sequencer, being the `Orchestrator`.

4.4.3. Transaction handling

As we're dealing with object-based Move, the transactions must specify the objects it wants to touch and a list of operations on those objects. These objects could be one of the following:

- Immutable objects. These objects never change. They stay on the chain forever. Published smart contracts fall in this category.
- Owned objects. These are objects owned by an address. The address is the only one capable of utilizing this object. When this object is used in a transaction, it will be consumed and if not deleted, their next version will be created. If this object is accessed at the same time in different transactions, it's called a double spend.
- Shared objects. These objects can be referenced by multiple transactions at the same time. The sequencing algorithm decides in what order they will be executed.

Unlike in UTXO-based systems, the transactions must be executed to know their exact outcome. We will not use a custom transaction and object format so that we can be compatible with the SUI tooling.

Transaction Pipeline

When an **owned object** is used in a transaction, it also carries with it its object version. Each time the object is used in an execution of a transaction, the version increases. At any point in time, only one transaction can consume this object at a specific version. Therefore, we can say that owned objects are implicitly ordered. We do, however, still need to deal with double spends of these objects. In the case of SUI, if there is a double spend detected, the system will lock this owned object until the next epoch. We assume that double spends of owned objects are only user errors, and no explicit fraudulent behavior is intended. When a user submits multiple transactions with the same owned object as input, the only harm it could potentially do is to itself. Would a user try to front-run its own transaction? The only thing a double spender could achieve is a cancelation of its previously submitted transaction, by locking this owned object until the next epoch, where it is freed automatically.

The same applies for **immutable objects**, but for these objects the version never changes. So we do not have to worry about transaction ordering for these type of objects.

The type of objects that do need to be taken care of are the **shared objects**. These need to be ordered before execution.

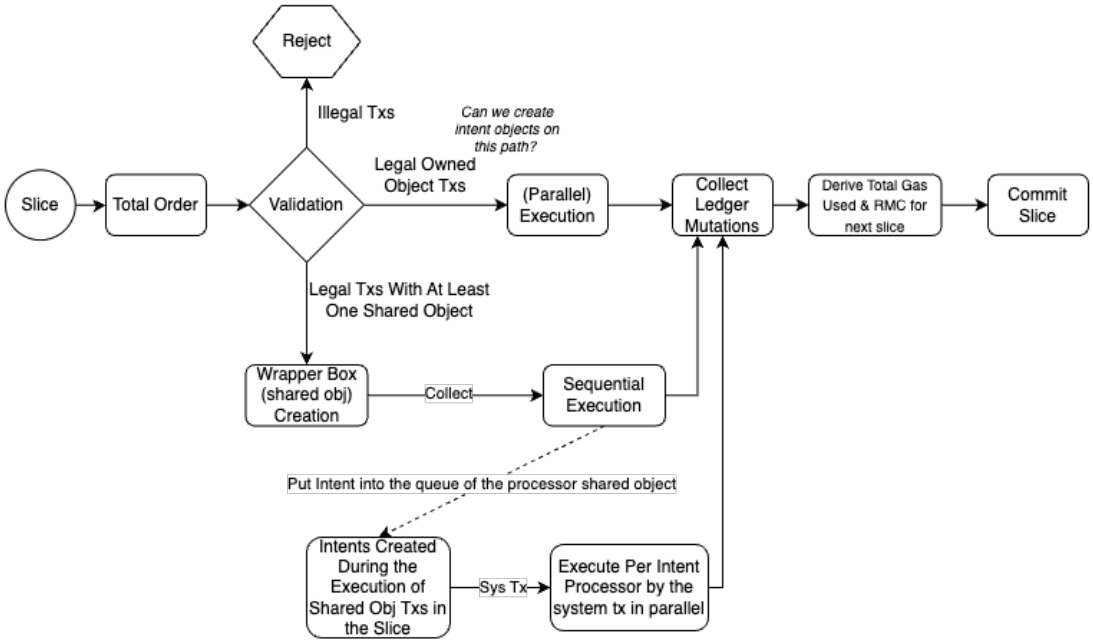


Figure 4.3: Execution pipeline

The full execution pipeline begins with a slice that is composed by the consensus slicing phase. This slice is a chunk of the DAG, containing blocks of transactions. Each block contains one transaction block, which is similar to a PTB in SUI and a Script in Aptos. This transaction block can contain multiple Move function calls, for example, a simple token transfer function call.

1. The given slice first has to be totally ordered. This is to detect and resolve conflicts. Conflicts are double spends: transactions that consume the same owned object with the same version.
2. Before execution, the transaction must be validated. This process involves:
   - Check if the signatures are valid.
   - Check if the gas object contains enough MANA to pay for the gas budget of the transaction.
   - Check if the gas budget is within the minimum and maximum limits.
   - Check if object-owned objects have the root object in the input list of the transaction.
   - Check if the owned-objects resolve to the transaction sender.
   - Check if the objects referenced in the transaction input exist in the current ledger state. Owned objects can be checked whether their current version and digest matches the one that was given in as input. Shared objects can be checked simply by checking if they exist with the given object ID. We do not try to match their version, as we do not know which version they will interact with before execution. The sequencer decides this later on in the pipeline.
3. Transactions not passing the validation phase are counted as Illegal transactions, and are rejected. These transactions are simply not executed. They do not cause any change in the ledger state. These transactions do still pay for block dissemination and pay a flat fee for validity checks.
4. Transactions that do pass the validation phase and that only contain owned objects and/or immutable objects can be executed in parallel. This is due to there being no overlap in transaction objects.
5. For the transactions passing the validation phase and that touch shared objects are wrapped in a `WrapperBox` alongside all the owned and/or immutable objects that they reference. The execution is then postponed and collected till the validation of all transactions in the slice has been concluded.

6. Once all the WrapperBoxes are generated, they will be transferred to their respective sequencer. Initially, and in this thesis, there will only be one default sequencer, the Orchestrator.
7. The orchestrator is a Move module that does the ordering of the WrapperBoxes and executes them sequentially. The default order for now will be the order in which the transactions have been validated. This will be done only for normal transactions that touch shared objects. The user also has the ability to send an intent transaction.
8. This intent transaction is similar to a normal transaction, but only signals to the Move module that it can use its custom orchestrator to process this transaction. A custom orchestrator could include pre- and post-processing steps surrounding the execution of the transactions. For example, a DEX could implement their own custom orchestrator in which it executes multiple transactions in less transactions, saving gas fees. The transactions are essentially batched like this.
9. All intents that are found during the sequential execution phase are placed into the queue of their respective orchestrator. These could be multiple different transactions in multiple orchestrator queues.
10. A system transaction, being a transaction initiated by the system itself, calls a function of the orchestrator which processes and executes the transactions in their queue. This can be done in parallel.
11. After all transactions have been executed, their TransactionEffects are collected in a list.
12. From that list, the total used gas is derived and the Reference Mana Cost (RMC) is adjusted for the next slice.
13. Finally, the list of TransactionEffects will be committed to the ledger state.

Transaction pipeline example

To make this process a bit more clear we will give an example of the whole process.

1. **Ledger state before processing a slice**
   The current ledger state at the end of the previous slice is like the one illustrated in 4.4:

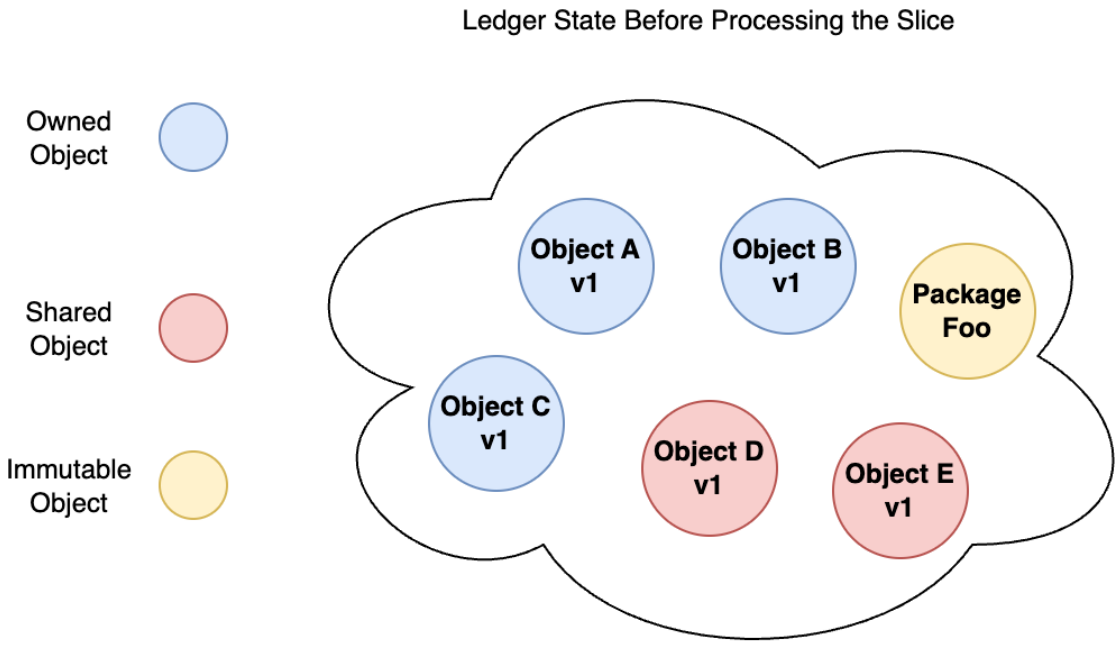


Figure 4.4: Initial Ledger State Example

This state consists of owned objects, shared objects and an immutable package which contains

Move modules. Each object besides the package has a version number. The shared objects also require a version number which is being used internally by the node itself. A transaction does not need to know that version number, only their object ID.

2. **Preparing the slice**
   The tangle is sliced by a leader block. The result of this slicing is a slice, which is visualized in Figure 4.5

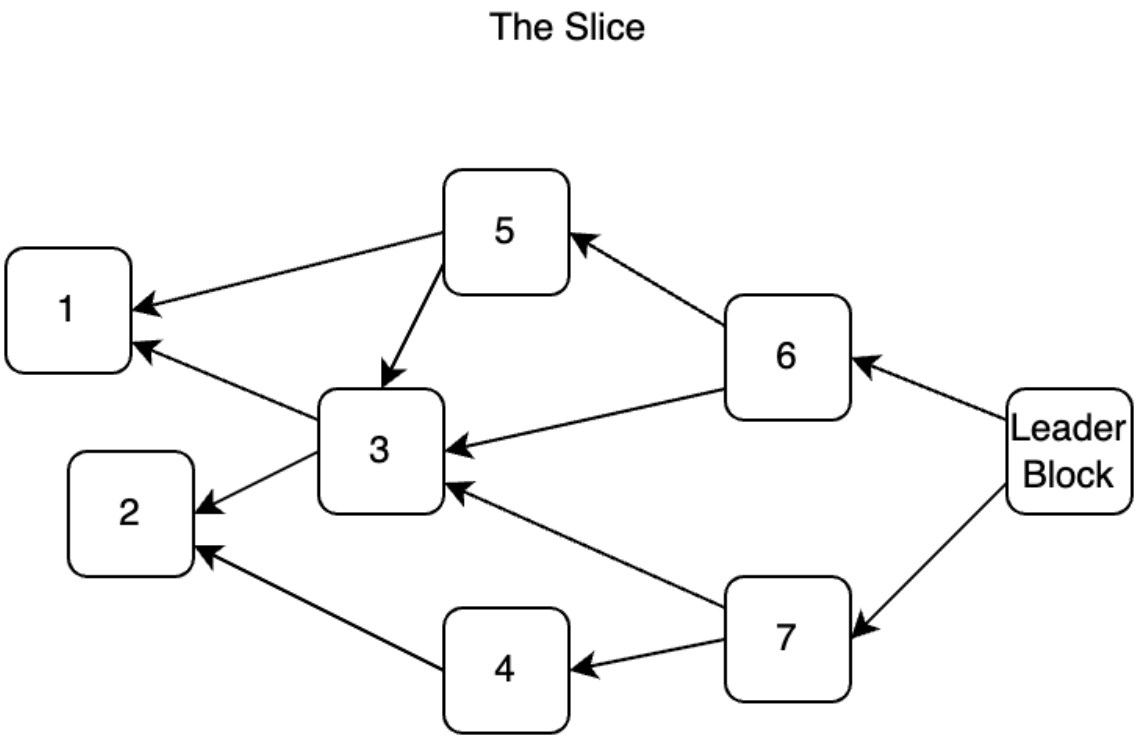


Figure 4.5: The Slice Example

For simplicity sake, we assume that each block contains only one transaction. This transaction is numbered after their block, so Block 1 contains Transaction 1. Block 1 and block 2 reference blocks that are in the previous slice.

3. **Ordering with WhiteFlag algorithm**
   WhiteFlag is an algorithm used for ordering the Tangle. It uses a post-order Depth-First-Search (DFS) algorithm, which starts at the milestone block. In this case, it will start from the leader block. The given slice in the previous step (Figure 4.5) will be ordered with this algorithm as shown in Figure 4.6.

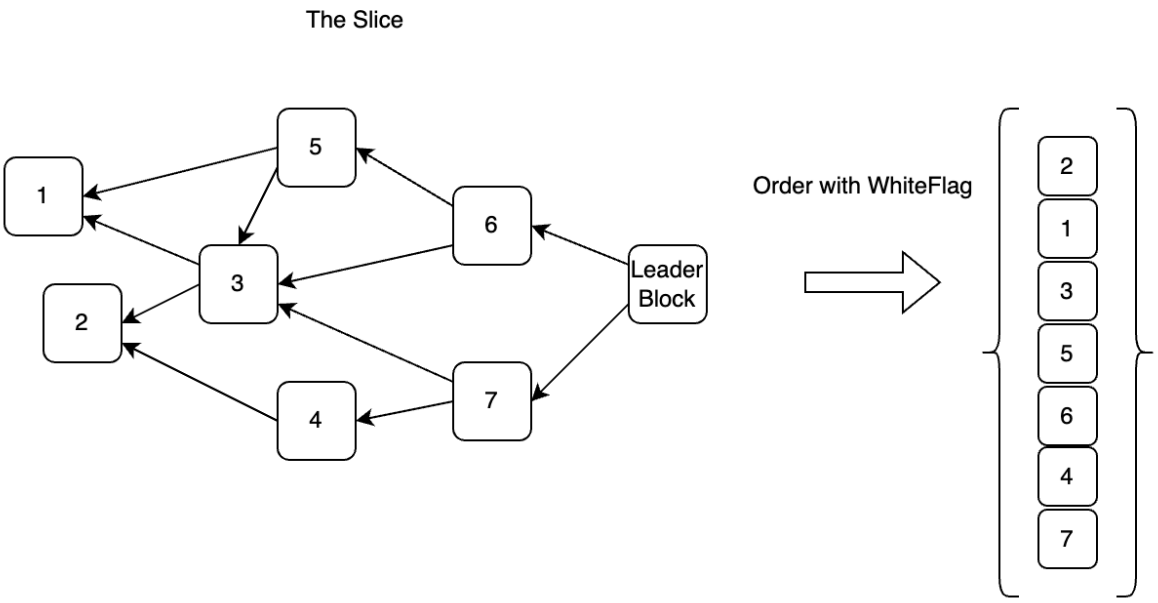


Figure 4.6: Slice ordered with WhiteFlag

The pseudo-algorithm for the algorithm can be found in Appendix F Listing F.1

As this algorithm is already used in the current production version of IOTA, we can preserve this algorithm and order for determining the order of the transactions in the slice. As previously mentioned, other ordering algorithms are also possible, but this seems to be the easiest to get the transactions in the slice totally ordered, which is required for the validation step.

4. **Validation**
   After the slice has been ordered, it will be validated. For this validation, we must have access to the current ledger state. If a transaction uses an owned object, it must be locked for further use in that slice. Any other transaction that does use that owned object in the slice will be seen as an illegal transaction. This validation process is done sequentially, since that way we are able to detect any double spends. The actual execution of owned objects can be parallelized.

   In Figure 4.7, we see the order that was determined in the previous step. Transaction 2 will be validated first. As this transaction only has owned and immutable objects as input (all objects that it touches), it can be executed directly. The gas payment object A v1 (version 1) is locked after validation of Transaction 2 is done. The validation of Transaction 1 does not need to wait for the result of the execution of Transaction 2. Since Transaction 1 uses the same gas object as Transaction 2, which is now locked, it will not pass validation and is seen as an illegal transaction. Consequently, the transaction is rejected and will not be passed forward in the pipeline.

   Figure 4.8 shows the continuation of the validation process.

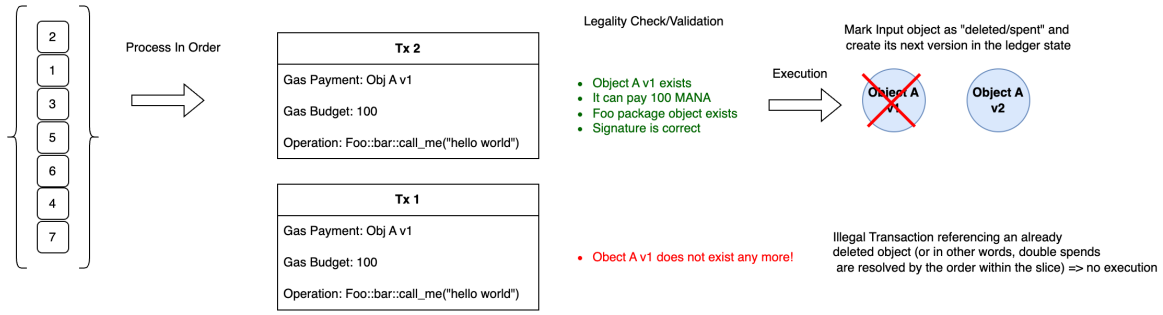


Figure 4.7: Validation of Slice

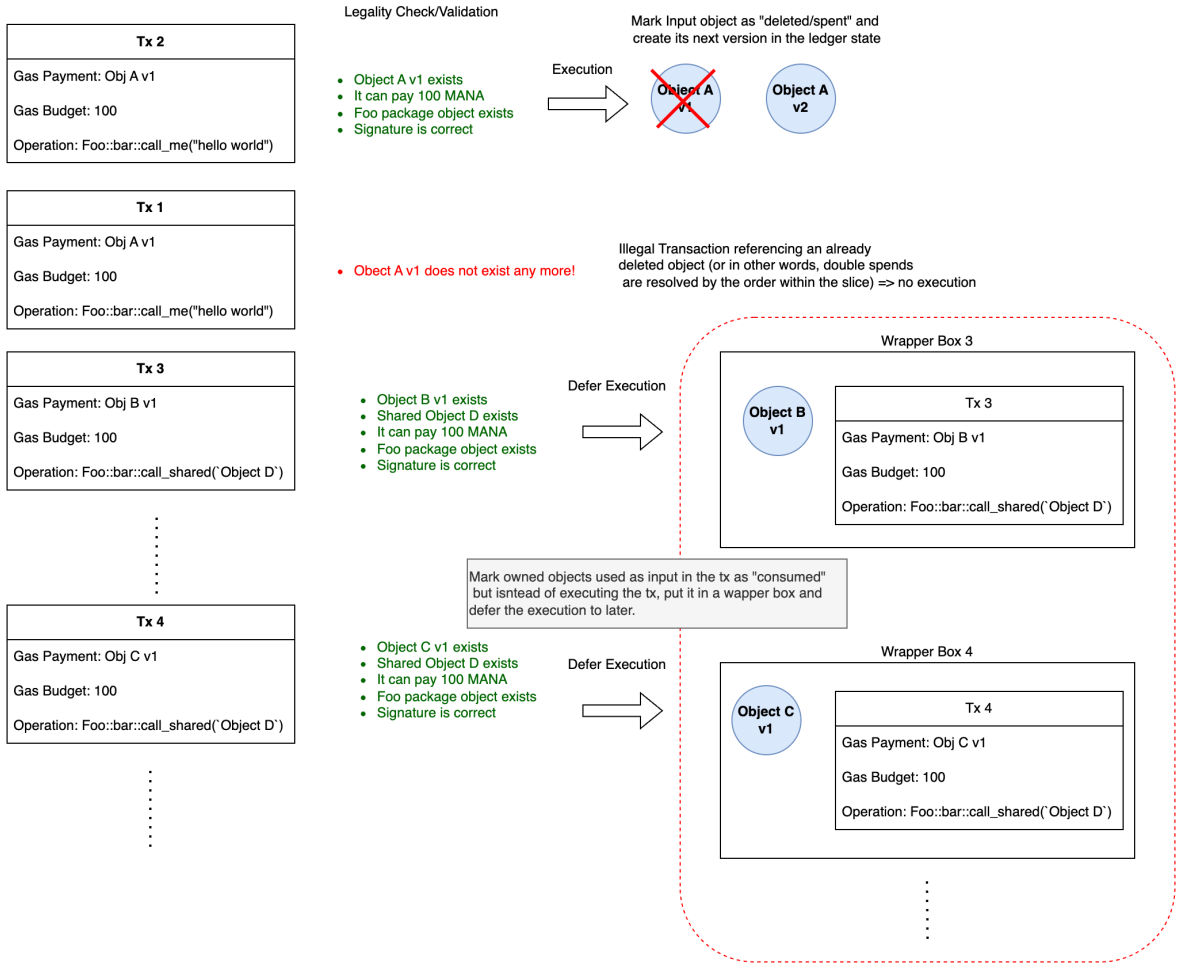


Figure 4.8: Validation of Slice (continued)

Transactions 3 and 4 touch the same shared object, but have different gas objects, so they pass the validation. But since they touch the same shared object in the same slice, they have to be ordered. The ordering, or sequencing, will be prepared by wrapping them in so called Wrapper Boxes. The execution of these transactions is deferred to a later stage. Once all the transactions in the slice are validated, all the Wrapper Boxes are collected and ready for the next stage. Each box has a label which states its Orchestrator ID. This dictates which sequencer will process them. Each shared object has this ID upon its creation. A transaction that tries to send a shared object with a different Orchestrator ID than its own, will be invalidated by the validation stage and be

deemed illegal. For the first implementation in this thesis, we will only have a default Orchestrator.

5. **Orchestration/Execution**
   The orchestration phase begins by passing the list of collected boxes to the Orchestrator Move module. The orchestrator chooses an order in which it will execute the transactions. This could for example be any of the previously mentioned ordering algorithms. For this implementation, we will preserve the WhiteFlag order. Transactions that touch different shared objects might be executed in parallel, but that will not be implemented for this current design. Thus, the implementation will only support sequential execution. In Figure 4.9, Transaction 4 will execute with the next version of Object D, as it has been ordered after the execution of Transaction 3.

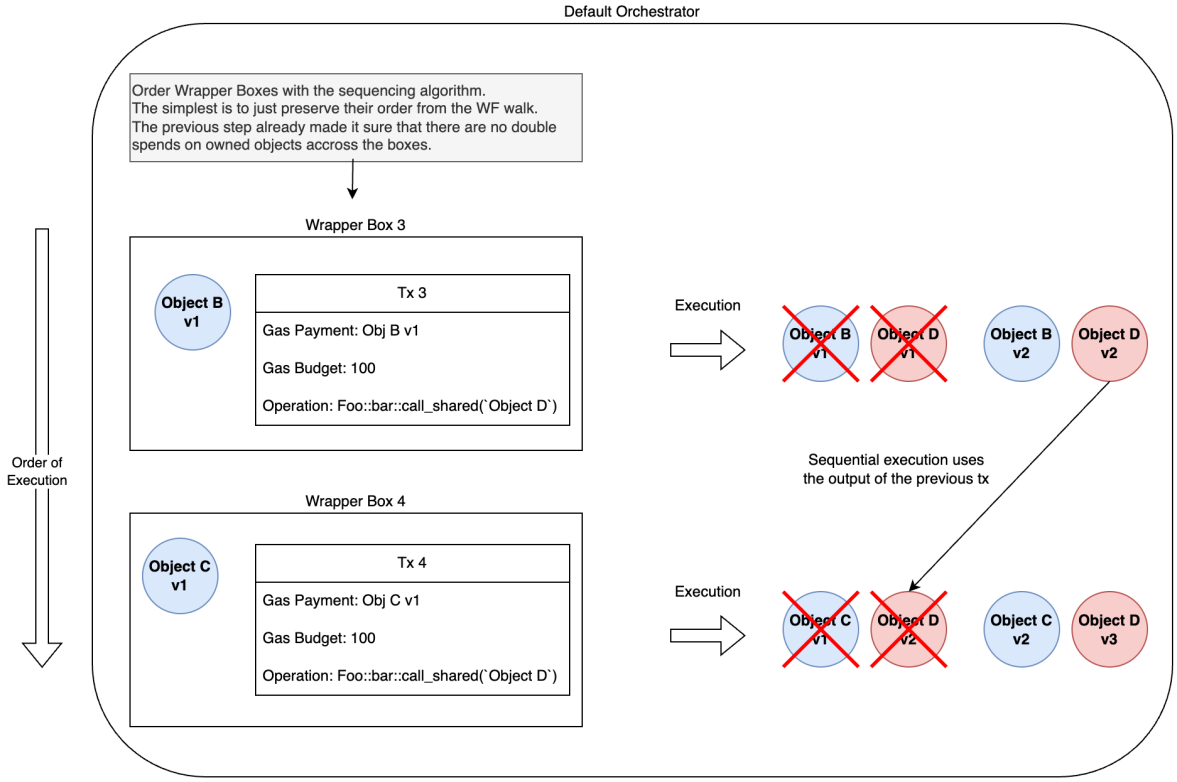


Figure 4.9: Orchestration/Execution step

6. **Collection of Ledger Mutations**
   Now that all the transactions have been executed, all the ledger mutations of both Wrapper Boxes and Owned objects, also known as their TransactionEffects, are collected and committed to the ledger.

   Before committing, we count the consumed execution gas from the TransactionEffects and adjust the RMC for the next slice accordingly. This is done to combat excessive computational resource usage.

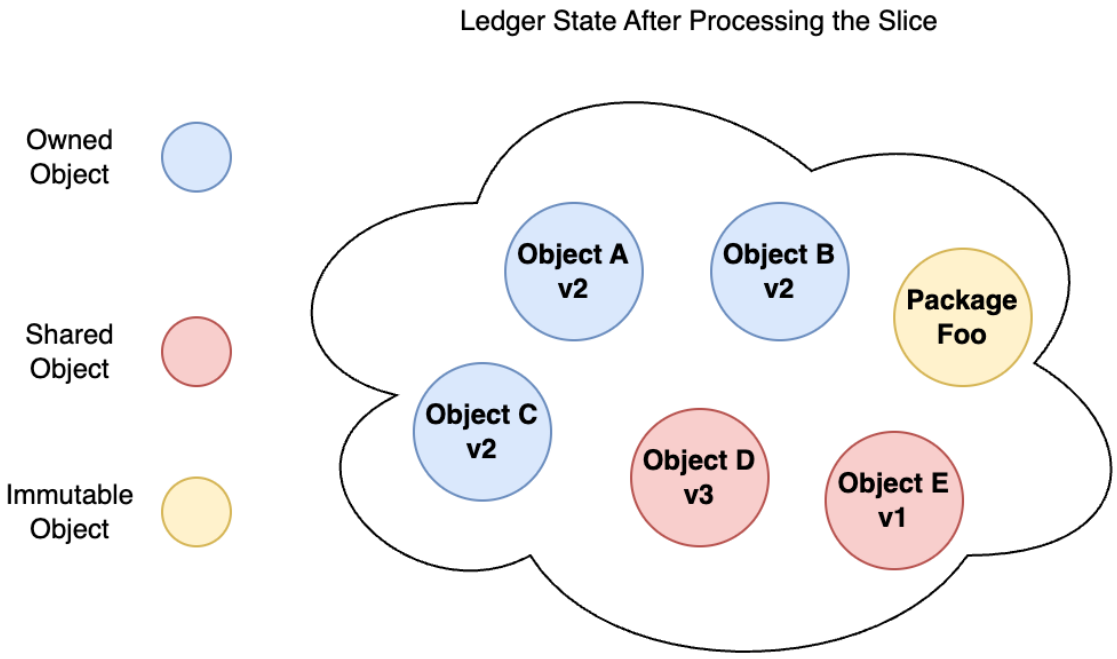


Figure 4.10: Ledger State After Slice is Procesesed

4.4.4. Custom sequencer

The custom sequencer will be explained a bit more in detail. The functionality of the custom sequencer will be:

- The custom sequencer will be able to have a pre-process function in the module, which is called by the system before any other transaction. Instead of being a separate function, it could also be implemented in a post-process function, where the last part of that function can be the pre-processor for the next round.
- It will also be able to have a post-process function in the module, which is called by the system through a system transaction. This function will be called at last, after all transactions in the queue have been executed.
- The custom sequencer will also be able to pay for the gas for these transactions, which could be by taking the gas object from a transaction.
- The module needs to be able to give feedback to the user, such as errors that occurred.
- The developer needs to be able to define the actual implementation logic of pre- and post-processing functions.

An example of such a custom processor module can be found in the Appendix F Listing F.2.

And an example of an implementation using the module can subsequently be found in F.3 of the same Appendix.

5 Evaluation

5.1. Testing the new system

To ensure correct functionality of the new system, we created ten scenarios to test the MoveVM White-Flag algorithm which modifies the ledger state. These scenarios involve transactions that resemble real world interactions that could also result in conflicts in the object ledger. These scenarios include creation, modification and deletion of objects, as well as wrapping and handling dynamic fields within objects. For each of the scenarios we start with an initial state of the ledger, which is loaded into Hornet. We then run the necessary transactions and compare the resulted ledger state against the expected end state of the ledger. The environment is an isolated environment, such that no other transactions are being executed, other than the ones specified in the specific scenario.

The ten scenarios are defined as such:

1. **publish_call_simple_package:** This scenario first publishes a simple move package with a single entry function, and then calls the entry function.
2. **create_owned_objects:** This scenario creates a three new objects through a single transaction calls.
3. **delete_owned_objects:** This scenario deletes a single owned object already present in the starting ledger state.
4. **dynamic_fields:** This scenario adds three dynamic fields, and then reads and removes them through three successive transactions that rely on the `working_bench` package. In addition, we generate three partial scenarios for each gradual state transition. That is, from genesis to the added fields, from the added fields to the read fields, and from the read fields to the removed fields.
5. **dynamic_object_fields:** This scenario adds three dynamic objects fields, and then reads and removes them through three successive transactions that rely on the `working_bench` package. In addition, we generate three partial scenarios for each gradual state transition. That is, from genesis to the added fields, from the added fields to the read fields, and from the read fields to the removed fields.
6. **wrap_object:** This scenario creates two single owned objects already present in the starting ledger state and wraps one of them into the other.
7. **unwrap_object:** This scenario unwraps a `working_bench::tools::Simple object` from a `working_bench::tools::SimpleWrapper` object. This filled `SimpleWrapper` object is already present in the starting ledger state.
8. **unwrap_and_delete_object:** This scenario unwraps and deletes a working_bench::tools::Simple object from a `working_bench::tools::SimpleWrapper` object. The filled `SimpleWrapper` object is already present in the starting ledger state.

9. **abort:** This scenario aborts the execution of a transaction due to insufficient gas. The only object that is written is the mutated gas coin owned by the caller.
10. **shared_object:** In this scenario, the Whiteflag algorithm is tested with multiple transactions that attempt to mutate a shared object. This scenario makes use of the `DonutBox`, a shared object introduced in the `working_bench::donuts` module. A `DonutBox` can contain a fixed number of Donuts. A user can try to get a `Donut` from the `DonutBox` by calling the function `donuts::get_donut` function if there are any Donuts left. The scenario is intended to create a competing situation in which two users simultaneously attempt to retrieve the last `Donut` from the `DonutBox` and only one of them can be successful. This scenario defines 4 different users, each with their own address and gas coin. Furthermore, we define the `DonutBox` with a capacity of 3 donuts, leading to a competition for the last donut.

Each of the scenarios contains a script that produces a json file which contains a list of ”VersionedObjectID : ObjectBytes”-pairs representing the starting ledger state, a list of transactions to execute for that specific scenario, and an expected end ledger state with the same format as the starting state.

With this json file, the ledger state is initialized with each of the transaction as a block on the sub-tangle.

Finally, a forged milestone is issued to trigger the WhiteFlag algorithm, which updates the ledger state. This new ledger state is then compared against the expected ledger state specified in the json file.

5.1.1. Publish Call Simple Package

This scenario showcases the ability to publish a package and then subsequently call an entry function that was in the published package.

The starting ledger state consists solely of the default genesis objects and two gas coins sent to the publisher of the package and caller of the function.

The two transactions that are executed are as follows:

1. publish_tx: The transaction to publish the package, sent and signed by the publisher.
2. call_tx: The transaction to call the package function `call_me`, sent and signed by the caller.

The sub tangle that is initialized is visualized in figure 5.1. As seen clearly in the figure, the publish transaction should be executed first, and then the call transaction. The forged milestone should finally reference the call transaction, after which the WhiteFlag algorithm modifies the ledger state.

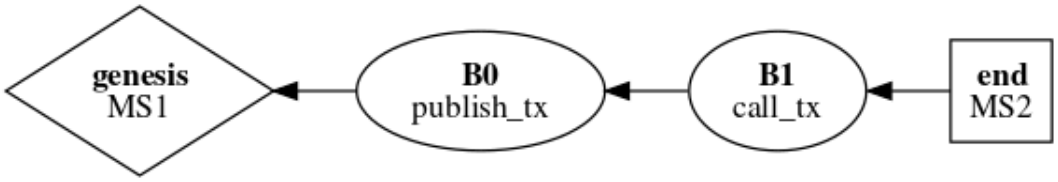


Figure 5.1: Tangle representation of the publish-call scenario

The expected end state of the ledger is:

- **Written**: We expect four new objects:
  1. A move package for the simple package.
  2. An `InitCap` object created during initialization of the package.
  3. The mutated gas coin owned by the publisher.
  4. The mutated gas coin owned by the caller.
- **Deleted**: We expect two outdated objects:
  1. The gas coin used in `publish_tx`.
  2. The gas coin use in `call_tx`.

This scenario has been successfully tested, as the end ledger state was the same as the expected ledger state.

5.1.2. Other Scenarios

The rest of the scenarios are also successfully tested, and are described similarly in Appendix G.

5.1.3. Results

The testing of the WhiteFlag algorithm across ten diverse scenarios has been key in demonstrating the feasibility of integrating MoveVM with the IOTA ledger. Each scenario was carefully designed to cover a wide range of potential interactions and conflicts that could arise in a real-world deployment. The successful execution and validation of these scenarios provide concrete evidence that the integration is not merely theoretical but operationally viable.

The scenarios were specifically chosen to challenge the system's ability to handle a wide range of transactions involving object creation, modification, deletion, and dynamic field manipulation. Most notably, the shared_object scenario tested the system's sequencing of transactions on shared resources.

The results from each scenario showed that the modified WhiteFlag algorithm correctly updated the ledger state as expected. This was verified by comparing the resulting ledger state after executing the transactions with the predefined expected state. The success of these tests proves that the WhiteFlag algorithm can effectively sequence Move transactions and interact with the results produced by MoveVM.

5.2. Cost Efficiency Batch Swap Intent Processor

This section will show the evaluation of the cost efficiency of the `batch_swap` intent-based module, which is designed to optimize the swap execution of buy and sell orders. The designed approach is economically beneficial because it first attempts to directly match buy and sell intents, thereby bypassing the need to go through the liquidity pool. If there are no matches, the function will use the pool for the remaining swaps.

To assess the cost efficiency of the module, we use a predefined set of swap transactions on two different scenarios:

1. Scenario 1: Execute the set of transactions without batch swap.
    - In this scenario, all transactions are processed directly through the liquidity pool without any attempt to match intents.
    - This approach can lead to higher costs due to pool usage costs, including slippage and fees.
2. Scenario 2: Execute the set of transactions with batch swap.
    - In this scenario, the batch swap functionality will first attempt to match buy and sell intents directly. Only the remaining unmatched intents are processed through the pool.
    - It is expected that this approach is more cost-effective due to the reduction of transactions that utilize the pool, lowering overall fees and slippage costs.

The sets of transactions are as follows:

- Set 1:
    1. Swap 10 USD for MANA
    2. Swap 10 MANA for USD
- Set 2:
    1. Swap 10 USD for MANA
    2. Swap 10 MANA for USD
    3. Swap 20 USD for MANA
    4. Swap 10 MANA for USD

The pool is initialized with 100 MANA and 100 USD liquidity. Fee of the pool is set to 0 for the evaluation. This means that the price is 1:1, meaning 1 MANA equals 1 USD.

All these transactions will be submitted within the same milestone, and will therefore be processed within the same milestone cone.

5.2.1. Results

The metrics used in the results as follows:

- Transaction Fees: The cost associated with executing a transaction.
- Slippage: The difference between the expected price of a trade and the actual price. This difference will be taken based on the actual current price from the pool without price impact consideration.
- Total Cost: Sum of transaction fees and slippage.

We calculate the slippage by comparing the expected output amount with the actual output amount. This difference in price is used in the following formulas to determine the slippage:

$$\text{Expected Output Amount} = \frac{\text{Amount Used to Swap}}{\text{Initial Price}} \tag{5.1}$$

$$\text{Slippage} = \text{Expected Output Amount} - \text{Actual Output Amount} \tag{5.2}$$

$$\text{Slippage}\% = \left(\frac{\text{Slippage}}{\text{Expected Output Amount}}\right) \times 100 \tag{5.3}$$

Table 5.1: Transaction Costs for Each Scenario, Set 1

Transaction	Scenario 1 (Normal Swap)			Scenario 2 (Batch Swap)		
	Fee	Slippage	Output	Fee	Slippage	Output
Swap 10 USD for MANA	3.22×10^{-6}	9.1	9.1	4.73×10^{-6}	0	10
Swap 10 MANA for USD	3.22×10^{-6}	-9.0	10.9	4.73×10^{-6}	0	10

The pool has not been used. Pool liquidity is still 100 MANA and 100 USD.

Table 5.2: Transaction Costs for Each Scenario, Set 2

Transaction	Scenario 1 (Normal Swap)			Scenario 2 (Batch Swap)		
	Fee	Slippage	Output	Fee	Slippage	Output
Swap 10 USD for MANA	3.22×10^{-6}	9.1	9.1	4.73×10^{-6}	0	10
Swap 10 MANA for USD	3.22×10^{-6}	-9.0	10.9	4.73×10^{-6}	0	10
Swap 20 USD for MANA	1.61×10^{-6}	15.3	16.9	4.73×10^{-6}	4.5	19.1
Swap 10 MANA for USD	1.61×10^{-6}	-26.8	12.7	4.73×10^{-6}	0	10

The pool has been used once, to swap the remaining 10 MANA for USD. Pool liquidity is 90.9 USD and 110 MANA.

It should be noted that in our prototype, the actual transaction fees of the system transaction that triggers the post-process function of the intent module are mitigated, as the system pays for it. In other words, the validators sponsor these transactions. In a real scenario, someone would need to pay for these transactions, which in a normal non-intent based situation would be the user. Therefore, a more realistic approach would be that the user also provides an extra gas-fee object which is used to pay for the gas fees. As these can differ based on the state of the intent module, excess gas fees can be refunded by the post-process function.

Based on the results shown in Tables 5.1 and 5.2, it is apparent that using the Batch Swap intent-based functionality gives a more stable swap output, relative to the original intent of the user. It effectively reduces overall slippage for the users, and therefore improves the cost-effectiveness and user experience for the user.

6

Conclusion

This thesis demonstrated that the modular execution environment MoveVM can be successfully used on top of IOTA for layer-1 programmability. This adaptation bridged two different blockchain ecosystems and technologies, and showed the versatility and potential of Move in various blockchain environments.

The Sui flavor of the MoveVM seemed to be the most fitting for IOTA, and its implementation was shown in two phases. In the first phase, a prototype of MoveVM was developed with a mock IOTA node, showcasing the potential and understanding of the new technology. In the second phase, this implementation was integrated with the current functional IOTA node software, resulting in a fully functional layer-1 smart contract execution platform for IOTA.

The basic functionality of this new system has been proven by testing the WhiteFlag algorithm for various scenarios, as this algorithm handles the transactions and therefore the MoveVM interactions.

Furthermore, an application-layer Move module has been designed and built on top of this new system as a practical application, and to introduce further enhancements to the new system on IOTA.

The integration of native on-chain intent execution in L1 smart contracts introduces significant potential for enhancing user experience through abstraction and simplification. Chapter 4 presented the design and implementation of a prototype application-layer intent module and proposed a design of a more comprehensive intent-based execution pipeline.

Intents represent user desires for state transitions without specifying exact execution paths. This approach involves solvers that construct transactions by matching user intents with optimal execution paths. The simple intent-based architecture facilitates automatic execution of lined-up intents, optimizing transaction efficiency. For instance, a batch-swap functionality in a DEX can execute trades in the most cost-effective manner.

The design and implementation of the simple intent-based architecture focused on a swap module that allows token exchanges between MANA and USD. The evaluation demonstrated the economic benefits of the batch swap intent processor by reducing the costs associated with slippage. The results highlighted that the intent-based approach significantly enhances cost efficiency and stability compared to traditional transaction processing.

Ironically, the concept of intents aims to abstract and simplify the user experience. However, this added simplicity opens up a range of new possibilities and technological freedoms. For example, with such an advanced intent-based architecture, fully-fledged custom on-chain VMs become possible, paving the way for innovative applications and enhanced functionality in DLTs.

References

[1] Satoshi Nakamoto. “Bitcoin: A Peer-to-Peer Electronic Cash System”. In: (2008).

[2] Serguei Popov. “The tangle”. In: *White paper* 1.3 (2018), p. 30.

[3] Leemon Baird. “Hashgraph consensus: fair, fast, byzantine fault tolerance”. In: *Swirlds Tech Report, Tech. Rep.* (2016).

[4] Bitcoin.org. *Running A Full Node*. Accessed: 2023-11-16. n.d. URL: https://bitcoin.org/en/full-node.

[5] Tim Congdon. “What Were the Causes of the Great Recession? The Mainstream Approach vs the Monetary Interpretation”. In: *World Economics* 15.2 (Apr. 2014), pp. 45–65.

[6] Barry M Leiner et al. *A brief history of the Internet*. ACM SIGCOMM Computer Communication Review, 2009.

[7] David Flanagan. *JavaScript: The Definitive Guide*. ”O’Reilly Media, Inc.”, 2006.

[8] Hillman Curtis Houston. *Flash Web Design: The art of motion graphics*. ”New Riders”, 1999.

[9] John Resig and Bear Bibeault. *Secrets of the JavaScript Ninja*. ”Manning Publications”, 2013.

[10] Nick Szabo. “Smart contracts: Building blocks for digital markets”. In: *Extropy* 16.18 (1997), p. 28.

[11] CoinGecko. *CoinGecko: Cryptocurrency Prices and Market Capitalization*. 2024. URL: https://www.coingecko.com/ (visited on 02/21/2024).

[12] Mahendra Shrivas and Dr Yeboah. “The Disruptive Blockchain: Types, Platforms and Applications”. In: (Dec. 2018). DOI: 10.21522/TIJAR.2014.SE.19.02.Art003.

[13] Vitalik Buterin, Gavin Wood, and Ethereum Project. *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform*. 2013. URL: https://ethereum.org/en/whitepaper/.

[14] DefiLlama. *EVM Chains Overview*. 2024. URL: https://defillama.com/chains/EVM (visited on 06/20/2024).

[15] Ethereum Foundation. *Token Standards*. 2024. URL: https://ethereum.org/en/developers/docs/standards/tokens/ (visited on 12/20/2023).

[16] Onchain Times. *The Rise of Alternative Virtual Machines (altVMs)*. 2024. URL: https://www.onchaintimes.com/p/the-rise-of-alternative-virtual-machines (visited on 05/15/2024).

[17] Manuel MT Chakravarty et al. “The extended UTXO model”. In: *Financial Cryptography and Data Security: FC 2020 International Workshops, AsiaUSEC, CoDeFi, VOTING, and WTSC, Kota Kinabalu, Malaysia, February 14, 2020, Revised Selected Papers 24*. Springer. 2020, pp. 525–539.

[18] Manuel Chakravarty et al. *Functional blockchain contracts*. 2019.

[19] Fuel Network. *What is Fuel?* 2024. URL: https://docs.fuel.network/docs/intro/what-is-fuel/ (visited on 06/20/2024).

[20] Radix DLT. *What is Radix Engine?* 2024. URL: https://learn.radixdlt.com/article/what-is-radix-engine (visited on 06/20/2024).

[21] S. Blackshear et al. “Move: A Language With Programmable Resources”. In: (2019). Revised September 25th, 2019.

[22] Sam Blackshear. *Sam Blackshear on the Origins of Move*. 2024. URL: https://blog.sui.io/move-origins-sam-blackshear/ (visited on 06/20/2024).

[23] Jean-Yves Girard. “Linear logic”. In: *Theoretical computer science* 50.1 (1987), pp. 1–101.

[24] The Rust Programming Language. *References and Borrowing*. 2024. URL: https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html (visited on 01/07/2024).

[25] Fabian Vogelsteller and Vitalik Buterin. *EIP-20: ERC-20 Token Standard*. 2015. URL: https://eips.ethereum.org/EIPS/eip-20 (visited on 02/05/2024).

[26] Oualid Zaazaa and Hanan El Bakkali. "A systematic literature review of undiscovered vulnerabilities and tools in smart contract technology". In: *Journal of Intelligent Systems* 32.1 (2023), p. 20230038.

[27] Move Language Contributors. *Abilities*. 2024. URL: https://github.com/move-language/move/blob/548f632ca3ace2707c59c96dda198a9a7bead592/language/changes/3-abilities.md (visited on 02/05/2024).

[28] Jingyi Emma Zhong et al. "The move prover". In: *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I 32*. Springer. 2020, pp. 137–150.

[29] K Rustan M Leino. "This is boogie 2". In: *manuscript KRML* 178.131 (2008), p. 9.

[30] Move Language Contributors. *Specification Language for Move Prover*. 2024. URL: https://github.com/move-language/move/blob/main/language/move-prover/doc/user/spec-lang.md (visited on 06/20/2024).

[31] Flow. *Introduction to Cadence*. 2024. URL: https://cadence-lang.org/docs/ (visited on 06/20/2024).

[32] StarkWare. *Cairo Programming Language*. 2024. URL: https://www.cairo-lang.org/ (visited on 06/20/2024).

[33] Diem Association. *Move Programming Language*. 2022. URL: https://github.com/diem/move/tree/main/language (visited on 12/20/2023).

[34] Diem Association. *Move*. 2022. URL: https://github.com/move-language/move (visited on 12/26/2023).

[35] Movement. *Movement Documentation*. 2024. URL: https://movement.gitbook.io/movement/ (visited on 12/26/2023).

[36] Aptos Labs. *Cryptography in Move*. 2024. URL: https://aptos.dev/move/move-on-aptos/cryptography/ (visited on 04/26/2024).

[37] Mysten Labs. *Move Concepts*. 2024. URL: https://docs.sui.io/concepts/sui-move-concepts/ (visited on 04/26/2024).

[38] Rati Gelashvili et al. "Block-stm: Scaling blockchain execution by turning ordering curse to a performance blessing". In: *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 2023, pp. 232–244.

[39] Aptos Labs. *Block-STM: How We Execute Over 160k Transactions Per Second on the Aptos Blockchain*. 2024. URL: https://medium.com/aptoslabs/block-stm-how-we-execute-over-160k-transactions-per-second-on-the-aptos-blockchain-3b003657e4ba (visited on 04/27/2024).

[40] Sebastian Müller et al. "Reality-based UTXO ledger". In: *Distributed Ledger Technologies: Research and Practice* 2.3 (2023), pp. 1–33.

[41] IOTA Foundation. *Introduction to Stardust*. 2024. URL: https://wiki.iota.org/learn/protocols/stardust/introduction/ (visited on 01/10/2024).

[42] Sui Foundation. *Dynamic (Object) Fields*. 2024. URL: https://docs.sui.io/concepts/dynamic-fields (visited on 03/01/2024).

[43] Diem Foundation. *Binary Canonical Serialization (BCS)*. 2024. URL: https://github.com/diem/bcs (visited on 03/01/2024).

[44] The Go Authors. *The Go Programming Language*. GitHub. 2024. URL: https://github.com/golang/go (visited on 03/01/2024).

[45] Mozilla. *uniffi-rs: a multi-language bindings generator for Rust*. GitHub. 2024. URL: https://github.com/mozilla/uniffi-rs (visited on 01/21/2024).

[46] Thibault Martinez. *IOTA TIPS - TIP 0002 - White Flag Ordering*. https://github.com/iotaledger/tips/blob/main/tips/TIP-0002/tip-0002.md. Accessed: 2023-12-22. 2020.

[47] IOTA Foundation. *IOTA Node Extension interface definitions*. GitHub. 2024. URL: `https://github.com/iotaledger/inx` (visited on 02/22/2024).

[48] Christopher Goes, Awa Sun Yin, and Adrian Brink. “Anoma: Undefining Money”. In: (2021).

A

Move Bytecode Table

Opcode	Bytecode
0x01	`Bytecode::Pop`
0x02	`Bytecode::Ret`
0x03	`Bytecode::BrTrue(offset)`
0x04	`Bytecode::BrFalse(offset)`
0x05	`Bytecode::Branch(offset)`
0x06	`Bytecode::LdU64(int_const)`
0x07	`Bytecode::LdConst(idx)`
0x08	`Bytecode::LdTrue`
0x09	`Bytecode::LdFalse`
0x0A	`Bytecode::CopyLoc(idx)`
0x0B	`Bytecode::MoveLoc(idx)`
0x0C	`Bytecode::StLoc(idx)`
0x0D	`Bytecode::MutBorrowLoc(idx)`
0x0E	`Bytecode::ImmBorrowLoc(idx)`
0x0F	`Bytecode::MutBorrowField(fh_idx)`
0x10	`Bytecode::ImmBorrowField(fh_idx)`
0x11	`Bytecode::Call(idx)`
0x12	`Bytecode::Pack(sd_idx)`
0x13	`Bytecode::Unpack(_sd_idx)`
0x14	`Bytecode::ReadRef`
0x15	`Bytecode::WriteRef`
0x16	`Bytecode::Add`
0x17	`Bytecode::Sub`
0x18	`Bytecode::Mul`
0x19	`Bytecode::Mod`
0x1A	`Bytecode::Div`
0x1B	`Bytecode::BitOr`
0x1C	`Bytecode::BitAnd`
0x1D	`Bytecode::Xor`
0x1E	`Bytecode::Or`
0x1F	`Bytecode::And`
0x20	`Bytecode::Not`
0x21	`Bytecode::Eq`
0x22	`Bytecode::Neq`
0x23	`Bytecode::Lt`
0x24	`Bytecode::Gt`
0x25	`Bytecode::Le`

0x26	`Bytecode::Ge`
0x27	`Bytecode::Abort`
0x28	`Bytecode::Nop`
0x29	`Bytecode::Exists(sd_idx)`
0x2A	`Bytecode::MutBorrowGlobal(sd_idx)`
0x2B	`Bytecode::ImmBorrowGlobal(sd_idx)`
0x2C	`Bytecode::MoveFrom(sd_idx)`
0x2D	`Bytecode::MoveTo(sd_idx)`
0x2E	`Bytecode::FreezeRef`
0x2F	`Bytecode::Shl`
0x30	`Bytecode::Shr`
0x31	`Bytecode::LdU8`
0x32	`Bytecode::LdU128`
0x33	`Bytecode::CastU8`
0x34	`Bytecode::CastU64`
0x35	`Bytecode::CastU128`
0x36	`Bytecode::MutBorrowFieldGeneric(fi_idx)`
0x37	`Bytecode::ImmBorrowFieldGeneric(fi_idx)`
0x38	`Bytecode::CallGeneric`
0x39	`Bytecode::PackGeneric`
0x3A	`Bytecode::UnpackGeneric(_si_idx)`
0x3B	`Bytecode::ExistsGeneric(si_idx)`
0x3C	`Bytecode::MutBorrowGlobalGeneric(si_idx)`
0x3D	`Bytecode::ImmBorrowGlobalGeneric(si_idx)`
0x3E	`Bytecode::MoveFromGeneric(si_idx)`
0x3F	`Bytecode::MoveToGeneric(si_idx)`
0x40	`Bytecode::VecPack(si, num)`
0x41	`Bytecode::VecLen(si)`
0x42	`Bytecode::VecImmBorrow(si)`
0x43	`Bytecode::VecMutBorrow(si)`
0x44	`Bytecode::VecPushBack(si)`
0x45	`Bytecode::VecPopBack(si)`
0x46	`Bytecode::VecUnpack(si, num)`
0x47	`Bytecode::VecSwap(si)`
0x48	`Bytecode::LdU16`
0x49	`Bytecode::LdU32`
0x4A	`Bytecode::LdU256`
0x4B	`Bytecode::CastU16`
0x4C	`Bytecode::CastU32`
0x4D	`Bytecode::CastU256`

B

batch_swap package source code

```
"""
swap.move
"""

module batch_swap::swap {
    use iota::vec_map::VecMap;
    use iota::vec_map;
    use iota::bag;
    use iota::bag::Bag;
    use iota::mana::MANA;
    use iota::object::{Self, UID, ID};
    use iota::coin::{Self, Coin};
    use iota::balance::{Self, Supply, Balance};
    use iota::transfer;
    use iota::math;
    use iota::tx_context::{Self, TxContext};
    use batch_swap::usd::{USD};
    use iota::event;

    /// For when supplied Coin is zero.
    const EZeroAmount: u64 = 0;

    /// For when pool fee is set incorrectly.
    /// Allowed values are: [0-10000).
    const EWrongFee: u64 = 1;

    /// For when someone tries to swap in an empty pool.
    const EReservesEmpty: u64 = 2;

    /// For when initial LSP amount is zero.
    const EShareEmpty: u64 = 3;

    /// For when someone attempts to add more liquidity than u128 Math allows.
    const EPoolFull: u64 = 4;

    /// The integer scaling setting for fees calculation.
    const FEE_SCALING: u128 = 10000;

    /// The max value that can be held in one of the Balances of
    /// a Pool. U64 MAX / FEE_SCALING
    const MAX_POOL_VALUE: u64 = {
        18446744073709551615 / 10000
    };

    /// The Pool token that will be used to mark the pool share
    /// of a liquidity provider. The first type parameter stands
    /// for the witness type of a pool. The seconds is for the
    /// coin held in the pool.
    struct LSP has drop {}
```

```
50
51      /// The pool with exchange.
52      ///
53      /// - `fee_percent` should be in the range: [0-10000), meaning
54      /// that 1000 is 100% and 1 is 0.1%
55      struct Pool has key {
56          id: UID,
57          mana: Balance<MANA>,
58          token: Balance<USD>,
59          lsp_supply: Supply<LSP>,
60          /// Fee Percent is denominated in basis points.
61          fee_percent: u64
62      }
63
64      /// Create new `Pool` for token `T`. Each Pool holds a `Coin<T>`
65      /// and a `Coin<MANA>`. Swaps are available in both directions.
66      ///
67      /// Share is calculated based on Uniswap's constant product formula:
68      ///  liquidity = sqrt( X * Y )
69      public entry fun create_pool(
70          token: Coin<USD>,
71          mana: Coin<MANA>,
72          fee_percent: u64,
73          ctx: &mut TxContext
74      ) {
75          let mana_amt = coin::value(&mana);
76          let tok_amt = coin::value(&token);
77
78          assert!(mana_amt > 0 && tok_amt > 0, EZeroAmount);
79          assert!(mana_amt < MAX_POOL_VALUE && tok_amt < MAX_POOL_VALUE, EPoolFull);
80          assert!(fee_percent >= 0 && fee_percent < 10000, EWrongFee);
81
82          // Initial share of LSP is the sqrt(a) * sqrt(b)
83          let share = math::sqrt(mana_amt) * math::sqrt(tok_amt);
84          let lsp_supply = balance::create_supply(LSP {});
85          let lsp = balance::increase_supply(&mut lsp_supply, share);
86
87          transfer::share_object(Pool {
88              id: object::new(ctx),
89              token: coin::into_balance(token),
90              mana: coin::into_balance(mana),
91              lsp_supply,
92              fee_percent
93          });
94
95          transfer::public_transfer(coin::from_balance(lsp, ctx), tx_context::sender(ctx));
96      }
97
98
99      /// Entrypoint for the `swap_mana` method. Sends swapped token
100     /// to sender.
101     entry fun swap_mana_(
102         pool: &mut Pool, mana: Coin<MANA>, ctx: &mut TxContext
103     ) {
104         transfer::public_transfer(
105             swap_mana(pool, mana, ctx),
106             tx_context::sender(ctx)
107         )
108     }
109
110     /// Swap `Coin<MANA>` for the `Coin<T>`.
111     /// Returns Coin<T>.
112     public fun swap_mana(
113         pool: &mut Pool, mana: Coin<MANA>, ctx: &mut TxContext
114     ): Coin<USD> {
115         assert!(coin::value(&mana) > 0, EZeroAmount);
116
117         let mana_balance = coin::into_balance(mana);
118
119         // Calculate the output amount - fee
120         let (mana_reserve, token_reserve, _) = get_amounts(pool);
```

```
121
122         assert!(mana_reserve > 0 && token_reserve > 0, EReservesEmpty);
123
124         let output_amount = get_input_price(
125             balance::value(&mana_balance),
126             mana_reserve,
127             token_reserve,
128             pool.fee_percent
129         );
130
131         balance::join(&mut pool.mana, mana_balance);
132         coin::take(&mut pool.token, output_amount, ctx)
133     }
134
135     /// Entry point for the `swap_token` method. Sends swapped MANA
136     /// to the sender.
137     entry fun swap_token_(
138         pool: &mut Pool, token: Coin<USD>, ctx: &mut TxContext
139     ) {
140         transfer::public_transfer(
141             swap_token(pool, token, ctx),
142             tx_context::sender(ctx)
143         )
144     }
145
146     /// Swap `Coin<T>` for the `Coin<MANA>`.
147     /// Returns the swapped `Coin<MANA>`.
148     public fun swap_token(
149         pool: &mut Pool, token: Coin<USD>, ctx: &mut TxContext
150     ): Coin<MANA> {
151         assert!(coin::value(&token) > 0, EZeroAmount);
152
153         let tok_balance = coin::into_balance(token);
154         let (mana_reserve, token_reserve, _) = get_amounts(pool);
155
156         assert!(mana_reserve > 0 && token_reserve > 0, EReservesEmpty);
157
158         let output_amount = get_input_price(
159             balance::value(&tok_balance),
160             token_reserve,
161             mana_reserve,
162             pool.fee_percent
163         );
164
165         balance::join(&mut pool.token, tok_balance);
166         coin::take(&mut pool.mana, output_amount, ctx)
167     }
168
169     /// Entrypoint for the `add_liquidity` method. Sends `Coin<LSP>` to
170     /// the transaction sender.
171     entry fun add_liquidity_(
172         pool: &mut Pool, mana: Coin<MANA>, token: Coin<USD>, ctx: &mut TxContext
173     ) {
174         transfer::public_transfer(
175             add_liquidity(pool, mana, token, ctx),
176             tx_context::sender(ctx)
177         );
178     }
179
180     /// Add liquidity to the `Pool`. Sender needs to provide both
181     /// `Coin<MANA>` and `Coin<T>`, and in exchange he gets `Coin<LSP>` -
182     /// liquidity provider tokens.
183     public fun add_liquidity(
184         pool: &mut Pool, mana: Coin<MANA>, token: Coin<USD>, ctx: &mut TxContext
185     ): Coin<LSP> {
186         assert!(coin::value(&mana) > 0, EZeroAmount);
187         assert!(coin::value(&token) > 0, EZeroAmount);
188
189         let mana_balance = coin::into_balance(mana);
190         let tok_balance = coin::into_balance(token);
191
```

```
        let (mana_amount, tok_amount, lsp_supply) = get_amounts(pool);

        let mana_added = balance::value(&mana_balance);
        let tok_added = balance::value(&tok_balance);
        let share_minted = math::min(
            (mana_added * lsp_supply) / mana_amount,
            (tok_added * lsp_supply) / tok_amount
        );

        let mana_amt = balance::join(&mut pool.mana, mana_balance);
        let tok_amt = balance::join(&mut pool.token, tok_balance);

        assert!(mana_amt < MAX_POOL_VALUE, EPoolFull);
        assert!(tok_amt < MAX_POOL_VALUE, EPoolFull);

        let balance = balance::increase_supply(&mut pool.lsp_supply, share_minted);
        coin::from_balance(balance, ctx)
    }

    /// Entrypoint for the `remove_liquidity` method. Transfers
    /// withdrawn assets to the sender.
    entry fun remove_liquidity_(
        pool: &mut Pool,
        lsp: Coin<LSP>,
        ctx: &mut TxContext
    ) {
        let (mana, token) = remove_liquidity(pool, lsp, ctx);
        let sender = tx_context::sender(ctx);

        transfer::public_transfer(mana, sender);
        transfer::public_transfer(token, sender);
    }

    /// Remove liquidity from the `Pool` by burning `Coin<LSP>`.
    /// Returns `Coin<T>` and `Coin<MANA>`.
    public fun remove_liquidity(
        pool: &mut Pool,
        lsp: Coin<LSP>,
        ctx: &mut TxContext
    ): (Coin<MANA>, Coin<USD>) {
        let lsp_amount = coin::value(&lsp);

        // If there's a non-empty LSP, we can
        assert!(lsp_amount > 0, EZeroAmount);

        let (mana_amt, tok_amt, lsp_supply) = get_amounts(pool);
        let mana_removed = (mana_amt * lsp_amount) / lsp_supply;
        let tok_removed = (tok_amt * lsp_amount) / lsp_supply;

        balance::decrease_supply(&mut pool.lsp_supply, coin::into_balance(lsp));

        (
            coin::take(&mut pool.mana, mana_removed, ctx),
            coin::take(&mut pool.token, tok_removed, ctx)
        )
    }

    /// Public getter for the price of MANA in token T.
    /// - How much MANA one will get if they send `to_sell` amount of T;
    public fun mana_price(pool: &Pool, to_sell: u64): u64 {
        let (mana_amt, tok_amt, _) = get_amounts(pool);
        get_input_price(to_sell, tok_amt, mana_amt, pool.fee_percent)
    }

    /// Public getter for the price of token T in MANA.
    /// - How much T one will get if they send `to_sell` amount of MANA;
    public fun token_price(pool: &Pool, to_sell: u64): u64 {
        let (mana_amt, tok_amt, _) = get_amounts(pool);
        get_input_price(to_sell, mana_amt, tok_amt, pool.fee_percent)
    }

```

```
263
264     /// Get most used values in a handy way:
265     /// - amount of MANA
266     /// - amount of token
267     /// - total supply of LSP
268     public fun get_amounts(pool: &Pool): (u64, u64, u64) {
269         (
270             balance::value(&pool.mana),
271             balance::value(&pool.token),
272             balance::supply_value(&pool.lsp_supply)
273         )
274     }
275
276     /// Calculate the output amount minus the fee - 0.3%
277     public fun get_input_price(
278         input_amount: u64, input_reserve: u64, output_reserve: u64, fee_percent: u64
279     ): u64 {
280         // up casts
281         let (
282             input_amount,
283             input_reserve,
284             output_reserve,
285             fee_percent
286         ) = (
287             (input_amount as u128),
288             (input_reserve as u128),
289             (output_reserve as u128),
290             (fee_percent as u128)
291         );
292
293         let input_amount_with_fee = input_amount * (FEE_SCALING - fee_percent);
294         let numerator = input_amount_with_fee * output_reserve;
295         let denominator = (input_reserve * FEE_SCALING) + input_amount_with_fee;
296
297         (numerator / denominator as u64)
298     }
299
300     ///
301     /// INTENT RELATED FUNCTIONS
302     ///
303
304     /// The intent itself. The T determines the swap direction.
305     struct Intent<phantom T> has key, store {
306         id: UID,
307         pool_id: ID,
308         coin: Coin<T>,
309         sender: address
310     }
311
312     /// The queue that holds the intents.
313     struct IntentQueue has key {
314         id: UID,
315         items: Bag
316     }
317
318     /// Event for when someone enqueued an intent.
319     struct IntentEnqueued has copy, drop {
320     }
321
322     /// Initialize the module.
323     fun init(ctx: &mut TxContext) {
324         let queue = init_queue(ctx);
325
326         transfer::share_object(queue);
327     }
328
329     /// Initialize the queue.
330     public fun init_queue(ctx: &mut TxContext): IntentQueue {
331         IntentQueue {
332             id: object::new(ctx),
333             items: bag::new(ctx),
```

```
        }
    }

    /// Add an intent to the queue.
    public entry fun add_intent<T>(
        queue: &mut IntentQueue,
        pool: &Pool,
        coin: Coin<T>,
        ctx: &mut TxContext
    ) {
        let intent = Intent {
            id: object::new(ctx),
            pool_id: object::id(pool),
            coin,
            sender: tx_context::sender(ctx)
        };

        let length = bag::length(&queue.items);
        bag::add(&mut queue.items, length, intent);

        event::emit(IntentEnqueued {});
    }


    /// Process the queue!
    /// The purpose of this specific process is to batch and match the intents, so that sometimes
    /// the buys and sells can be settled without the need to go through the pool.
    /// This prevents slippage and allows for more efficient swaps.
    public entry fun process_queue(
        queue: &mut IntentQueue,
        pool: &mut Pool,
        ctx: &mut TxContext
    ) {
        let length = bag::length(&queue.items);

        if (length == 0) {
            return
        };

        // Collections to remember intent senders and values.
        let mana_intents = vec_map::empty<address, u64>();
        let usd_intents = vec_map::empty<address, u64>();

        let mana_balance = balance::zero<MANA>();
        let usd_balance = balance::zero<USD>();

        // Loop through all bag items, so for `length` iterations.
        let i = 0;

        while (i < length) {
            // Get the item and its type.
            let is_usd = bag::contains_with_type<u64, Intent<USD>>(&queue.items, i);

            if (is_usd) {
                // If it's USD, then it's a buy MANA swap.
                process_intent_object<USD>(queue, &mut usd_intents, &mut usd_balance, i);
            } else {
                // Else it's MANA, so it's a buy USD swap.
                process_intent_object<MANA>(queue, &mut mana_intents, &mut mana_balance, i);
            };

            i = i + 1;
        };

        // Try to match intents directly without pool.
        if (vec_map::size(&mana_intents) > 0 && vec_map::size(&usd_intents) > 0) {
            match_swaps(pool, &mut mana_intents, &mut usd_intents, &mut mana_balance, &mut usd_balance, ctx)
        };

```

```
403
404         // If there are any balances left, swap them through the pool.
405         if (balance::value(&mana_balance) > 0) {
406             let total_mana = balance::value(&mana_balance);
407
408             // Swap MANA for USD.
409             let usd = swap_mana(pool, coin::from_balance(mana_balance, ctx), ctx);
410
411             // Calculate shares and distribute them.
412             process_intent_swaps(&mut usd, total_mana, &mut mana_intents, ctx);
413
414             // If there are any leftovers, send them to the pool.
415             balance::join(&mut pool.token, coin::into_balance(usd));
416         } else {
417             balance::destroy_zero(mana_balance);
418         };
419
420         if (balance::value(&usd_balance) > 0) {
421             let total_usd = balance::value(&usd_balance);
422
423             // Swap USD for MANA.
424             let mana = swap_token(pool, coin::from_balance(usd_balance, ctx), ctx);
425
426             process_intent_swaps(&mut mana, total_usd, &mut usd_intents, ctx);
427             balance::join(&mut pool.mana, coin::into_balance(mana));
428         } else {
429             balance::destroy_zero(usd_balance);
430         };
431     }
432
433     /// Internal function to process the intent objects in the queue.
434     fun process_intent_object<T>(queue: &mut IntentQueue, collected_intents: &mut vec_map::
         VecMap<address, u64>, balance: &mut Balance<T>, index: u64) {
435         // If it's USD, then it's a buy MANA swap.
436         let intent = bag::remove<u64, Intent<T>>(&mut queue.items, index);
437
438         let Intent<T> {
439             id,
440             pool_id: _pool_id,
441             coin,
442             sender
443         } = intent;
444
445         if (vec_map::contains(collected_intents, &sender)) {
446             let amount = vec_map::get_mut(collected_intents, &sender);
447             *amount = (*amount + coin::value(&coin));
448         } else {
449             vec_map::insert(collected_intents, sender, coin::value(&coin));
450         };
451
452         // Add the intent to the balance.
453         balance::join(balance, coin::into_balance(coin));
454         object::delete(id);
455     }
456
457     /// Internal function to calculate the individual shares of all intents and distribute
         them.
458     fun process_intent_swaps<T>(coin: &mut Coin<T>, total_balance: u64, collected_intents: &
         mut VecMap<address, u64>, ctx: &mut TxContext) {
459         // Calculate shares and distribute them.
460         while (vec_map::size(collected_intents) > 0) {
461             let (address, amount) = vec_map::pop(collected_intents);
462             let share = (((amount as u128) * (coin::value(coin) as u128)) / (total_balance as
                  u128) as u64);
463
464             // Transfer the share to the intent sender.
465             let b = coin::split(coin, share, ctx);
466             transfer::public_transfer(b, address);
467         };
468     }
469
```

```
    /// Match swaps directly without the need to go through the pool. Increases economical efficiency and prevents
    /// slippage. It is also possible to partially match the swaps, which will be the default behavior of this function
    /// to keep it simple. Uses the current price without considering the pool's price impact.
    fun match_swaps(pool: &Pool, mana_intents: &mut VecMap<address, u64>, usd_intents: &mut VecMap<address, u64>, mana_balance: &mut Balance<MANA>, usd_balance: &mut Balance<USD>, ctx: &mut TxContext) {
        // Loop through all mana intents and try to match them with USD intents.
        let i = 0;
        let mana_price = current_mana_price_in_usd(pool);
        let _usd_price = current_usd_price_in_mana(pool);

        while (i < vec_map::size(mana_intents)) {
            let (mana_key, mana_value) = vec_map::get_entry_by_idx_mut(mana_intents, i);
            // now we have the key= address and value = amount of mana
            // we want to calculate the amount of usd correlates to the amount of mana
            let correlating_usd_amount = (mana_price * *mana_value)/(FEE_SCALING as u64); // divide by precision factor 10^4
            let j = 0;
            while (*mana_value > 0 && j < vec_map::size(usd_intents)) {
                // Just take the USD intents until we have enough to match the mana intent.
                let (usd_key, usd_value) = vec_map::get_entry_by_idx_mut(usd_intents, j);
                if (*usd_value >= correlating_usd_amount) {
                    // We have enough USD to match the mana intent.

                    // Directly transfer the mana and usd balances.
                    let usd_bal = balance::split(usd_balance, correlating_usd_amount);
                    let mana_bal = balance::split(mana_balance, *mana_value);

                    transfer::public_transfer(coin::from_balance(usd_bal, ctx), *mana_key);
                    transfer::public_transfer(coin::from_balance(mana_bal, ctx), *usd_key);

                    // Reduce the mana and usd intents.
                    *mana_value = 0;
                    *usd_value = *usd_value - correlating_usd_amount;

                    // We actually don't want to remove any intents, only set their value to 0.
                    // This is because we're still iterating over them!

                    // // Remove the USD intent if it's empty.
                    // if (*usd_value == 0) {
                    //     vec_map::remove_entry_by_idx(usd_intents, j);
                    // };
                    //
                    // // Remove the MANA intent, as it's empty.
                    // vec_map::remove_entry_by_idx(mana_intents, i);
                } else if (*usd_value > 0) {
                    // We don't have enough USD to match the mana intent.

                    // Directly transfer the balances.
                    let usd_bal = balance::split(usd_balance, *usd_value);
                    let mana_bal = balance::split(mana_balance, (*usd_value * (FEE_SCALING as u64) / mana_price));

                    transfer::public_transfer(coin::from_balance(usd_bal, ctx), *mana_key);
                    transfer::public_transfer(coin::from_balance(mana_bal, ctx), *usd_key);

                    // Reduce the mana intent by the correlating USD amount.
                    *mana_value = *mana_value - (*usd_value * (FEE_SCALING as u64) / mana_price);
                    *usd_value = 0;

                    // // Remove the USD intent, as it's empty.
                    // vec_map::remove_entry_by_idx(usd_intents, j);
                };

                j = j + 1;
            };
```

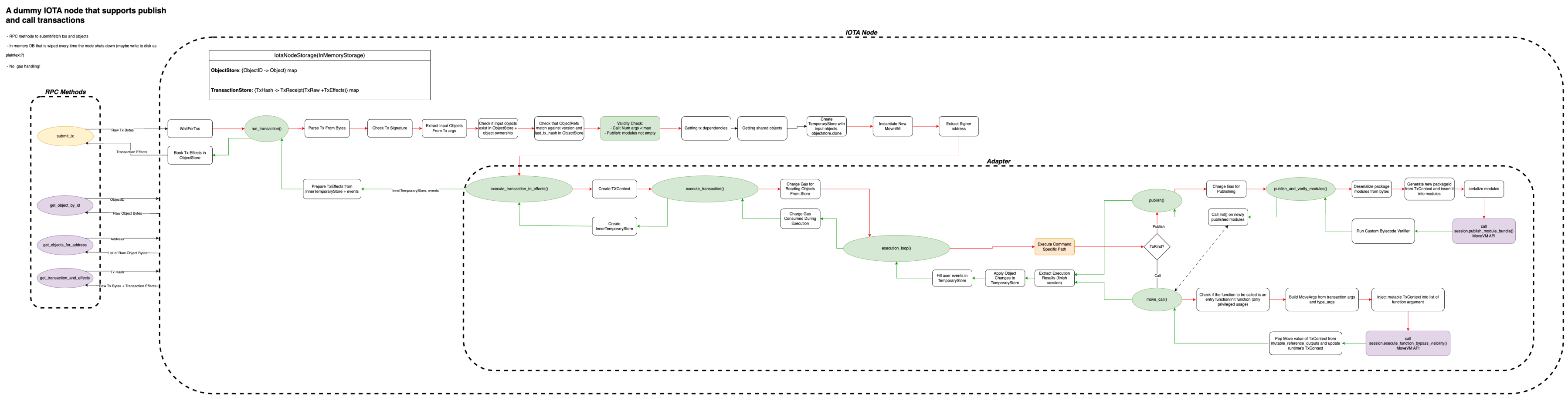
```
            i = i + 1;
        };
    }

    /// Get the current price of MANA in USD, without considering price fluctuations in the pool.
    public fun current_mana_price_in_usd(pool: &Pool): u64 {
        let (mana_amt, usd_amt, _) = get_amounts(pool);
        let (mana_amt, usd_amt) = ((mana_amt as u128), (usd_amt as u128));
        (((usd_amt * FEE_SCALING)/ mana_amt) as u64) // Price of 1 MANA in USD.
    }

    /// Get the current price of USD in MANA, without considering price fluctuations in the pool.
    public fun current_usd_price_in_mana(pool: &Pool): u64 {
        let (mana_amt, usd_amt, _) = get_amounts(pool);
        let (mana_amt, usd_amt) = ((mana_amt as u128), (usd_amt as u128));
        (((mana_amt * FEE_SCALING) / usd_amt) as u64) // Price of 1 USD in MANA.
    }

    #[test_only]
    /// Function to initialize the module for testing.
    public fun init_test(ctx: &mut TxContext) {
        init(ctx);
    }
}
```

```
"""
usd.move
"""

module batch_swap::usd {
    use std::option;
    use iota::tx_context::{Self, TxContext};
    use iota::transfer;
    use iota::coin;

    const EAlreadyMinted: u64 = 0;

    /// The total supply of Mana denominated in whole USD tokens (10 Billion)
    const TOTAL_SUPPLY_USD: u64 = 10_000_000_000;

    /// The total supply of Mana denominated in miniMana (10 Billion * 10^9)
    const TOTAL_SUPPLY_MINIUSD: u64 = 10_000_000_000_000_000_000;

    /// Name of the coin
    struct USD has drop {}

    /// Mint the new coin on init and send all to user.
    fun init(witness: USD, ctx: &mut TxContext) {
        let (treasury, metadata) = coin::create_currency(
            witness,
            9,
            b"USD",
            b"USD",
            b"USD EQUALS MONEY.",
            option::none(),
            ctx
        );
        transfer::public_freeze_object(metadata);
        let coin = coin::mint(&mut treasury, TOTAL_SUPPLY_MINIUSD, ctx);

        transfer::public_transfer(treasury, tx_context::sender(ctx));

        // Split the coin for ease of use in our swap dapp.
        let a = coin::split(&mut coin, 1_000_000_000, ctx);
        let b = coin::split(&mut coin, 1_000_000_000, ctx);
        let c = coin::split(&mut coin, 1_000_000_000, ctx);
        let d = coin::split(&mut coin, 1_000_000_000, ctx);

```

```
        transfer::public_transfer(coin, tx_context::sender(ctx));
        transfer::public_transfer(a, tx_context::sender(ctx));
        transfer::public_transfer(b, tx_context::sender(ctx));
        transfer::public_transfer(c, tx_context::sender(ctx));
        transfer::public_transfer(d, tx_context::sender(ctx));
    }
}
```

C

Detailled Execution Flow of Phase 1 Adapter



A dummy IOTA node that supports publish and call transactions
- RPC methods to submit/fetch txs and objects
- In memory DB that is wiped every time the node shuts down (maybe write to disk as plaintext?)
- No gas handling!
RPC Methods
submit_tx
Raw Tx Bytes
Transaction Effects
get_object_by_id
ObjectID
Raw Object Bytes
get_objects_for_address
Address
List of Raw Object Bytes
get_transaction_and_effects
Tx Hash
Raw Tx Bytes + Transaction Effects
IOTA Node
IotaNodeStorage(InMemoryStorage)
ObjectStore: {ObjectID -> Object} map
TransactionStore: {TxHash -> TxReceipt(TxRaw +TxEffects)} map
WaitForTxs
Book Tx Effects in ObjectStore
run_transaction()
Parse Tx From Bytes
Check Tx Signature
Extract Input Objects From Tx args
Check if Input objects exist in ObjectStore + object ownership
Check that ObjectRefs match against version and last_tx_hash in ObjectStore
Validity Check: - Call: Num args < max - Publish: modules not empty
Getting tx dependencies
Getting shared objects
Create TemporaryStore with input objects. objectstore.clone
Instantiate New MoveVM
Extract Signer address
Prepare TxEffects from InnerTemporaryStore + events
InnetTemporaryStore, events
Adapter
execute_transaction_to_effects()
Create TXContext
Create InnerTemporaryStore
execute_transaction()
Charge Gas for Reading Objects From Store
Charge Gas Consumed During Execution
execution_loop()
Fill user events in TemporaryStore
Apply Object Changes to TemporaryStore
Extract Execution Results (finish session)
Execute Command Specific Path
TxKind?
Publish
Call
publish()
Charge Gas for Publishing
publish_and_verify_modules()
Call Init() on newly published modules
Deserialize package modules from bytes
Generate new packageId from TxContext and insert it into modules
serialize modules
call session.publish_module_bundle() MoveVM API
Run Custom Bytecode Verifier
move_call()
Check if the function to be called is an entry function/init function (only privileged usage)
Build MoveArgs from transaction args and type_args
Inject mutable TxContext into list of function argument
call session.execute_function_bypass_visibility() MoveVM API
Pop Move value of TxContext from mutable_reference_outputs and update runtime's TxContext

D

TransactionEffectsV1 struct

```rust
pub struct TransactionEffectsV1 {
    /// The status of the execution
    pub status: ExecutionStatus,
    /// The epoch when this transaction was executed.
    pub milestone_index: u32,
    pub gas_used: GasCostSummary,
    /// The version that every modified (mutated or deleted) object had before
    /// it was modified by this transaction.
    pub modified_at_versions: Vec<(ObjectID, SequenceNumber)>,
    /// The object references of the shared objects used in this transaction.
    /// Empty if no shared objects were used.
    pub shared_objects: Vec<ObjectRef>,
    /// The transaction digest
    pub transaction_digest: TransactionDigest,
    /// ObjectRef and owner of new objects created.
    pub created: Vec<(ObjectRef, Owner)>,
    /// ObjectRef and owner of mutated objects, including gas object.
    pub mutated: Vec<(ObjectRef, Owner)>,
    /// ObjectRef and owner of objects that are unwrapped in this transaction.
    /// Unwrapped objects are objects that were wrapped into other objects in
    /// the past, and just got extracted out.
    pub unwrapped: Vec<(ObjectRef, Owner)>,
    /// Object Refs of objects now deleted (the old refs).
    pub deleted: Vec<ObjectRef>,
    /// Object refs of objects previously wrapped in other objects but now
    /// deleted.
    pub unwrapped_then_deleted: Vec<ObjectRef>,
    /// Object refs of objects now wrapped in other objects.
    pub wrapped: Vec<ObjectRef>,
    /// The updated gas object reference. Have a dedicated field for convenient
    /// access. It's also included in mutated.
    pub gas_object: (ObjectRef, Owner),
    /// The digest of the events emitted during execution,
    /// can be None if the transaction does not emit any event.
    pub events_digest: Option<TransactionEventsDigest>,
    /// The set of transaction digests this transaction depends on.
    pub dependencies: Vec<TransactionDigest>,
}
```

E

Phase 2 code blocks

Listing E.1: Timing Results UniFFI vs gRPC

```
//------------------------------ Uniffi ------------------------------//
=== RUN   TestScenarioPublishCallSimplePackage
2023/12/15 17:13:27 Uniffi execute call duration: 9.815492ms
2023/12/15 17:13:27 Uniffi execute call duration: 8.478777ms
2023/12/15 17:13:27 Uniffi execute call duration: 9.063965ms
2023/12/15 17:13:27 Uniffi execute call duration: 7.912293ms
--- PASS: TestScenarioPublishCallSimplePackage (0.56s)
PASS
ok      github.com/iotaledger/hornet-move/pkg/whiteflag/test    0.802s

=== RUN   TestScenarioPublishCallSimplePackage100
ok      github.com/iotaledger/hornet-move/pkg/whiteflag/test    55.898s

//------------------------------ gRPC ------------------------------//
=== RUN   TestScenarioPublishCallSimplePackage
2023/12/15 17:13:30 gRPC execute call duration: 6.497809ms
2023/12/15 17:13:30 gRPC execute call duration: 5.760803ms
2023/12/15 17:13:30 gRPC execute call duration: 5.485178ms
2023/12/15 17:13:30 gRPC execute call duration: 6.528254ms
--- PASS: TestScenarioPublishCallSimplePackage (0.55s)
PASS
ok      github.com/iotaledger/hornet-move/pkg/whiteflag/test    0.791s

=== RUN   TestScenarioPublishCallSimplePackage100
ok      github.com/iotaledger/hornet-move/pkg/whiteflag/test    55.223s
```

Listing E.2: ApplyConfirmationWithoutLocking function

```
func (m *Manager) ApplyConfirmationWithoutLocking(msIndex iotago.MilestoneIndex,
    written Objects, deleted DeletedObjects) error {
  ...
  // For each deleted object in confirmation, create a new Deleted Object in the
    storage and mark it as deleted in mutations
  for _, deletedObject := range deleted {
    if err := storeDeletedAndMarkObjectDeleted(deletedObject, mutations); err !=
    nil { ... }
    deletedObjMap[*deletedObject.VersionedObjectID()] = struct{}{}
  }
```

```
  // For each written object in confirmation, create a new Object in the storage
  for _, object := range written {
    if err := storeObject(object, mutations); err != nil { ... }
    // If it is not a written object that is also deleted, mark it as Alive
    if _, isDeleted := deletedObjMap[*object.VersionedObjectID()]; isDeleted {
    continue }
    if err := markAsAlive(object, mutations); err != nil { ... }
  }
  // Create the milestone diff
  msDiff := &MilestoneDiff{
    Index:   msIndex,
    Written: written,
    Deleted: deleted,
  }
  ...
}
```

Listing E.3: LegalTransactionResult message

```
message LegalTransactionResult {
    // The unique id of the transaction just executed.
    types.id.TransactionId transaction_id = 1;
    // This is the BCS form of a collection of lists of objects indexed by their
    ObjectReference that indicate the effects of the transaction execution; e.g.,
    lists of created objects, modified objects, etc. These are saved into the
    transaction's block metadata.
    bytes effects = 2;
    This is the BCS form of a collection of events thrown during the execution;
    these are saved into the transaction's block metadata.
    bytes events = 3;
    A list of ObjectReferences associated to the corresponding Move object encoded
     using the BCS form; these are the objects that were created, modified or
    unwrapped during the execution; these are saved into the object storage as
    AliveObjects;
    types.object.WrittenObjects written = 4;
    A list of ObjectReferences of the objects that were deleted or wrapped during
    the execution; these are used to mark the objects into the object storage as
    DeletedObjects;
    types.object.DeletedObjects deleted = 5;
}
```

Listing E.4: ConflictKind and ConflictError types

```
type ConflictKind uint8
const (
  ErrorNoConflict ConflictKind = iota
  ErrorInvalidSignature
  ErrorTransactionDeserializationError
  ...
)
type ConflictError struct {
  // The error kind.
  Kind ConflictKind
  // The error string message.
  Message string
  // The specific error object
  ErrorDetails ConflictErrorDetail
}
```

Old Code

```
type Output struct {
  outputID          iotago.OutputID
  blockID           iotago.BlockID
  msIndexBooked     iotago.MilestoneIndex
  msTimestampBooked uint32

  outputData []byte
  outputOnce sync.Once
  output     iotago.Output
}
```

New Code

```
// Object is a ledger state object on node level.
// It wraps the Move Object and additional metadata.
type Object struct {
  // the ID of the object
  objectID iotago.ObjectID
  // the sequence number of the object (also called version)
  sequenceNumber iotago.SequenceNumber
  // the ID of the block that contained the transaction that created/updated the
      object
  blockID iotago.BlockID
  // the index of the milestone that created/updated the object
  msIndexBooked iotago.MilestoneIndex
  // the timestamp of the milestone that created/updated the object
  msTimestampBooked uint32

  // BCS serialized Move Object
  objectBCSData []byte
}
```

Figure E.1: Output model replaced by Object model.

Listing E.5: Read Object from Node Storage code

```
object, err := deps.ObjectManager.ReadObjectWithoutLocking(objectID)
...
alive, err := deps.ObjectManager.IsObjectAliveWithoutLocking(version)
...
if alive { ... }
...
deleted, err := deps.ObjectManager.ReadDeletedObjectWithoutLocking(*version)
```

Listing E.6: MilestoneDiff type

```
// It contains the result of multiple executed transactions.
type MilestoneDiff struct {
  // The index of the milestone.
  Index iotago.MilestoneIndex
  // The newly created objects with this diff.
  Written Objects
  // The deleted objects with this diff
  Deleted DeletedObjects
}
```

Listing E.7: New Transaction Methods

```
service INX {
```

Old Code

```
// Spent are already spent TXOs (transaction outputs).
type Spent struct {
  outputID iotago.OutputID
  // the ID of the transaction that spent the output
  transactionIDSpent iotago.TransactionID
  // the index of the milestone that spent the output
  msIndexSpent iotago.MilestoneIndex
  // the timestamp of the milestone that spent the output
  msTimestampSpent uint32

  output *Output
}
```

New Code

```
// Deleted objects are objects that were deleted by a milestone confirmation.
type Deleted struct {
  objectID       iotago.ObjectID
  sequenceNumber iotago.SequenceNumber
  // the ID of the transaction that deleted the object
  transactionIDDeleted iotago.MoveTransactionID
  // the index of the milestone that deleted the object
  msIndexDeleted iotago.MilestoneIndex
  // the timestamp of the milestone that deleted the object
  msTimestampDeleted uint32

  // the object itself
  object *Object
}
```

Figure E.2: Spent model replaced by Deleted model.

```
  ...

  // Transctions
  rpc ReadTransaction(types.id.TransactionId) returns (types.block.
    TransactionWithResults);
  rpc ReadEvents(types.id.TransactionId) returns (types.block.RawTransactionEvents
    );
  rpc ReadEffects(types.id.TransactionId) returns (types.block.
    RawTransactionEffects);
  rpc SubmitTransaction(types.transaction.RawTransaction) returns (types.block.
    BlockId);
  rpc DryRunTransaction(types.transaction.RawTransaction) returns (types.block.
    DryRunTransactionResults);
  rpc ListenToTransactions(NoParams) returns (stream types.block.
    TransactionWithResults);
  rpc ListenEvents(NoParams) returns (stream types.block.RawTransactionEvents);
  rpc ListenEffects(NoParams) returns (stream types.block.RawTransactionEffects);

  ...
}
message RawTransaction {
    bytes data = 1;
}
message RawTransactionEffects {
```

```
  bytes data = 1;
}
message RawTransactionEvents {
  bytes data = 1;
}
message TransactionResult {
  RawTransactionEffects transaction_effects = 1;
  RawTransactionEvents transaction_events = 2;
}
message TransactionWithResults {
  transaction.RawTransaction transaction = 1;
  TransactionResult transaction_result = 2;
}
```

Listing E.8: Pushing new written and deleted objects

```
for _, written := range mutations.WrittenObjects {
  writtenObjects = append(writtenObjects, written)
}
for _, deleted := range mutations.DeletedObjects {
  deletedObjects = append(deletedObjects, deleted)
}
if err = objectManager.ApplyConfirmationWithoutLocking(milestoneIndex,
    writtenObjects, deletedObjects); err != nil {
  return fmt.Errorf("confirmMilestone: object.ApplyConfirmation failed: %w", err)
}
```

Listing E.9: Updating metadata of all transactions

```
if err := forBlockMetadataWithBlockID(referencedBlock.BlockID, func(meta *storage.
    CachedMetadata) {
  if referencedBlock.IsTransaction {
    if referencedBlock.Conflict.Kind != iotago.ErrorNoConflict {
      // IllegalTx, doesn't have tx result and tx id
      meta.Metadata().SetConflictingTx(referencedBlock.Conflict)
    } else {
      // if it is not a conflict, the tx is executed and has results + id
      if err = meta.Metadata().SetTransactionResult(*referencedBlock.
    TransactionResult); err != nil {
        fmt.Println("confirmMilestone: SetTransactionResult failed: %w", err)
      }
      meta.Metadata().SetTransactionID(*referencedBlock.TransactionID)
    }
  } else { ... }
  ...
})
```

Listing E.10: Prune MilestoneDiff Function

```
func (m *Manager) PruneMilestoneIndexWithoutLocking(msIndex iotago.MilestoneIndex)
    error {
  diff, err := m.MilestoneDiffWithoutLocking(msIndex)
  ...
  for _, deleted := range diff.Deleted {
    if err := deleteObject(deleted.object, mutations); err != nil { ... }
    if err := removeDeleted(deleted, mutations); err != nil { ... }
  }
  if err := deleteDiff(msIndex, mutations); err != nil { ... }
```

```
  ...
}
```

Listing E.11: Prune Blocks Function

```
func (p *Manager) pruneBlocks(blockIDsToDeleteMap map[iotago.BlockID]struct{}) int {
  for blockID := range blockIDsToDeleteMap {
    cachedBlockMeta := p.storage.CachedBlockMetadataOrNil(blockID) // meta +1
    ...
    cachedBlockMeta.ConsumeMetadata(func(metadata *storagepkg.BlockMetadata) { // meta -1
      // Delete the reference in the parents
      for _, parent := range metadata.Parents() {
        p.storage.DeleteChild(parent, blockID)
      }
      p.storage.DeleteTransaction(*metadata.TransactionID())
    })
    p.storage.DeleteBlock(blockID)
  }
  return len(blockIDsToDeleteMap)
}
```

Old Code

```
Output:
=======
Key:
    UTXOStoreKeyPrefixOutput [1 byte] + iotago.OutputID [34 bytes]

Value:
    BlockID [32 bytes] + MilestoneIndex [4 bytes] + MilestoneTimestamp [4 bytes] +
         iotago.Output.Serialized() [1 byte type + X bytes]

Spent Output:
================
Key:
    UTXOStoreKeyPrefixSpent [1 byte] + iotago.OutputID [34 bytes]

Value:
    TargetTransactionID (iotago.TransactionID) [32 bytes] + ConfirmationIndex (
        iotago.MilestoneIndex) [4 bytes] + ConfirmationTimestamp [4 bytes]

Unspent Output:
===============
Key:
    UTXOStoreKeyPrefixUnspent [1 byte] + iotago.OutputID [34 bytes]

Value:
    Empty
*/
```

New Code

```
Object:
=======
Key:
    ObjectStoreKeyPrefixObject [1 byte] + iotago.ObjectID [32 bytes]+
        SequenceNumber [8 bytes]

Value:
    BlockID [32 bytes] + MilestoneIndex [4 bytes] + MilestoneTimestamp [4 bytes] +
         iotago.Object.Serialized() [X bytes]

Deleted Objects:
================
Key:
    ObjectStoreKeyPrefixDeleted [1 byte] + iotago.ObjectID [32 bytes] +
        SequenceNumber [8 bytes]

Value:
    TargetTransactionID (iotago.TransactionID) [32 bytes] + ConfirmationIndex (
        iotago.MilestoneIndex) [4 bytes] + ConfirmationTimestamp [4 bytes]

Alive Objects:
==============
Key:
    ObjectStoreKeyPrefixAlive [1 byte] + iotago.ObjectID [32 bytes]

Value:
    SequenceNumber [8 bytes]
*/
```

Figure E.3: Old UTXO Database replaced by New Object Database.

Old Code

```
// Transaction is a transaction with its inputs, outputs and unlocks.
type Transaction struct {
  // The transaction essence, respectively the transfer part of a Transaction.
  Essence *TransactionEssence
  // The unlocks defining the unlocking data for the inputs within the Essence.
  Unlocks Unlocks
}
// TransactionEssence is the essence part of a Transaction.
type TransactionEssence struct {
  // The network ID for which this essence is valid for.
  NetworkID NetworkID
  // The inputs of this transaction.
  Inputs Inputs `json:"inputs"`
  // The commitment to the referenced inputs.
  InputsCommitment InputsCommitment `json:"inputsCommitment"`
  // The outputs of this transaction.
  Outputs Outputs `json:"outputs"`
  // The optional embedded payload.
  Payload Payload `json:"payload"`
}
```

New Code

```
type MoveTransaction struct {
  BCSSerializedSenderSignedTx []byte
}
```

Figure E.4: Old Transaction Format replaced by New MoveTransaction format.

Old Code

```
service INX {
  ...

  // UTXO
  rpc ReadOutput(OutputId) returns (OutputResponse);
  rpc ReadUnspentOutputs(NoParams) returns (stream UnspentOutput);

  rpc ListenToLedgerUpdates(MilestoneRangeRequest) returns (stream LedgerUpdate);
  rpc ListenToTreasuryUpdates(MilestoneRangeRequest) returns (stream
      TreasuryUpdate);
  rpc ListenToMigrationReceipts(NoParams) returns (stream RawReceipt);

  ...
}

message LedgerUpdate {
  ...
  oneof op {
    Marker batch_marker = 1;
    LedgerSpent consumed = 2;
    LedgerOutput created = 3;
  }
}
message LedgerOutput {
  OutputId output_id = 1;
  BlockId blockId = 2;
  uint32 milestone_index_booked = 3;
  uint32 milestone_timestamp_booked = 4;
  RawOutput output = 5;
}
message LedgerSpent {
  LedgerOutput output = 1;
  TransactionId transaction_id_spent = 2;
  uint32 milestone_index_spent = 3;
  uint32 milestone_timestamp_spent = 4;
}
```

New Code

```
service INX {
  ...

  // Objects
  rpc ReadObject(types.id.ObjectId) returns (types.ledger.StorageObjectResponse);
  rpc ReadObjects(NoParams) returns (stream types.ledger.LedgerObject);

  rpc ListenToLedgerUpdates(types.milestone.MilestoneRangeRequest) returns (stream
       types.ledger.LedgerUpdate); // equivalent to ListenToDeletedObjects and
      ListenToWrittenObjects

  ...
}
message StorageObjectResponse {
  uint32 ledger_index = 1;
  oneof payload {
    StorageObject object = 2;
    StorageDeletedObject deleted_object = 3;
  }
}
message LedgerObject {
  uint32 ledgerIndex = 1;
  id.ObjectRef object_ref = 2;
  StorageObject object = 3;
}
message LedgerUpdate {
  ...
  oneof op {
    Marker batch_marker = 1;
    StorageDeletedObject deleted = 2;
    StorageObject written = 3;
```

F

Advanced Architecture Code

Listing F.1: Pseudocode WhiteFlag Algorithm

```
update_ledger_state(ledger, milestone, solid_entry_points) {
    s = new Stack()
    visited = new Set()

    s.push(milestone)

    while (!s.is_empty()) {
        curr = s.peek()
        next = null

        // Look for the first eligible parent that was not already visited
        for parent in curr.parents {
          if (!solid_entry_points.contains(parent) && !parent.confirmed && !
    visited.contains(parent)) {
            next = parent
            break
          }
        }

        // All parents have been visited, apply and visit the current message
        if next == null {
          ledger.apply(curr)
          visited.add(curr)
          s.pop()
        }
        // Otherwise, go to the parent
        else {
          s.push(next)
        }
    }
}
```

Listing F.2: Custom Processor Module

```
module custom_processor::custom_processor {

    use iota::table_vec::{Self, TableVec};
    use iota::tx_context::{Self, TxContext};
```

```
    use std::ascii::{Self, String};
    use iota::object::{Self, UID};
    use std::type_name;

    struct IntentQueue has key {
        id: UID,
        items: TableVec
    }

    struct SequencerUsedEvent has copy, drop {
        sender: address,
        module: String,
        type: String,
    }

    public fun init_queue(ctx: &mut TxContext): IntentQueue {
        IntentQueue {
            id: object::new(ctx),
            items: table_vec::empty(ctx),
        }
    }

    public fun get_queue(ctx: &mut TxContext): IntentQueue {
        // Some magic in the TxContext coming from the node ensures a set queue in preprocessing
        // is part of the context in postprocessing and other functions in that module so it can be used in postprocessing
        tx_context::current_queue(ctx)
    }

    public fun add_to_queue<T>(item: T, ctx: &mut TxContext) {
        let queue = get_queue(ctx);
        table_vec::push_back(queue.items, item);

        let typename = type_name::get<T>();

        event::emit(SequencerUsedEvent {
            sender: tx_context::sender(ctx),
            module: type_name::get_module(typename),
            type: type_name::into_string(typename),
        });
    }

    public fun queue_length(ctx: &mut TxContext): u64 {
        let queue = get_queue(ctx);
        table_vec::length(queue.items)
    }

    public fun queue_pop<T>(ctx: &mut TxContext): T {
        let queue = get_queue(ctx);
        table_vec::pop_back(queue.items)
    }
}
```

Listing F.3: Custom Processor Module Implementation

```
module custom_processor::example {
```

```
2
3       use custom_processor::custom_processor;
4       use iota::tx_context::{TxContext};
5       use std::vector;
6       use std::ascii::{Self, String};
7       use iota::object::{Self, UID};
8       use iota::vec_set::{Self};
9
10      struct ConcatIntent has store {
11          sender: address,
12          text: String,
13      }
14
15      struct Souvenir has key, store {
16          id: UID,
17          final_text: String
18      }
19
20      // System call, only called if a event is found for a Intent for this module
21      public fun preprocess(ctx: &mut TxContext) {
22          custom_processor::init_queue(ctx);
23      }
24      /// invocation of the intent processor by a sys tx
25      public fun postprocess(ctx: &mut TxContext) {
26          let queue = custom_processor::get_queue(ctx);
27
28          let final_string = String::from_ascii("");
29          let senders = vec_set::empty<address>();
30
31          while(custom_processor::queue_length(ctx) > 0) {
32              let item = custom_processor::queue_pop<ConcatIntent>(ctx);
33              // TODO: Maybe implement some sorting here?
34              // Move lacks some basic sorting functions though so would make the
        example complex...
35              final_string = String::append(final_string, String::from_ascii(" "));
36              final_string = String::append(final_string, item.text);
37
38              if(!vec_set::contains(senders, item.sender)) {
39                  senders.insert(item.sender);
40              }
41          }
42
43          while(vec_set::size(senders) > 0) {
44              let souvenir = Souvenir {
45                  id: object::new(ctx),
46                  final_text: final_string,
47              };
48
49              let sender = vec_set::pop(senders); // Simplified, function does not
        actually exist here
50
51              transfer::public_transfer(souvenir, sender);
52          }
53
54      }
55
```

```
    public entry fun queue_concat(text: String, ctx: &mut TxContext) {

        let item = ConcatIntent {
            sender: tx_context::sender(ctx),
            text: text,
        };

        // This call abstracts adding to the queue and emitting an event
        custom_processor::add_to_queue(item, ctx);
    }
}
```

G

WhiteFlag Testing Scenarios

G.1. Publish Call Simple Package

This scenario has been described in 5.1.1.

G.2. Create Owned Objects

This scenario creates three new objects in a single transaction call. The function it calls is in a module that is already deployed in the genesis state.

The starting genesis state consists of the default genesis objects, a specially deployed package, and a gas coin transferred to the caller of the function.

The transaction that is executed is the following:

1. call_tx: The transaction to call the package function tools::create_simple, sent and signed by the caller.

The sub tangle that is initialized is visualized in Figure G.1, where the call transaction simply depends on the genesis.

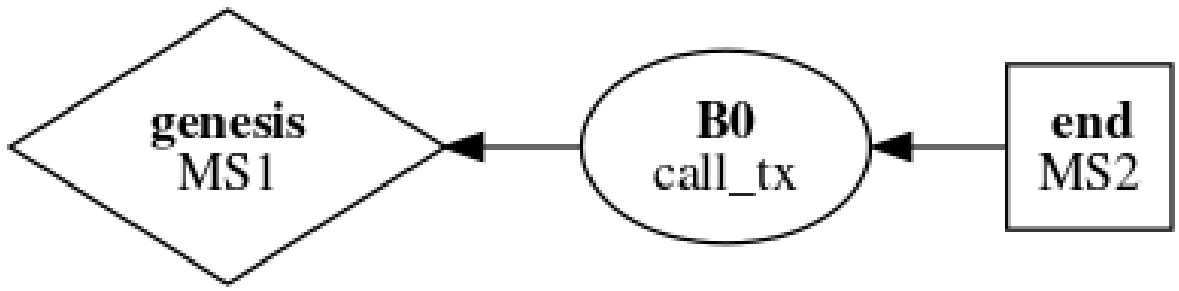


Figure G.1: Tangle representation of the create_owned_objects scenario.

The expected end state of the ledger is:

- **Written:** We expect four (4) new objects:
  1. The mutated gas coin owned by the caller.
  2. Three (3) working_bench::tools::Simple objects created through the transaction.
- **Deleted:** We expect one outdated object:
  1. The gas coin used in call_tx.

G.3. Delete Owned Objects

This scenario deletes a single owned object already present in the starting ledger state.

The transaction calls a package also present in the starting ledger state.

The starting genesis state consists of the default genesis objects, a specially deployed package, a gas coin transferred to the caller of the transaction, and three Simple objects.

The transaction that is executed is the following:

- call_tx: The transaction to call the package function tools::delete_simple, sent and signed by the caller.

The sub tangle that is initialized is similarly visualized as Figure G.1.

- **Written:** We expect one new objects:
  1. The mutated gas coin owned by the caller.
- **Deleted:** We expect two objects:
  1. The outdated gas coin used in call_tx.
  2. The deleted working_bench::tools::Simple object deleted through call_tx.

G.4. Dynamic Fields

This scenario adds three dynamic fields, and then reads and removes them through three successive transactions that rely on the working_bench package.

In addition we generate three partial scenarios for each gradual state transition. That is, from genesis to the added fields, from the added fields to the read fields, and from the read fields to the removed fields.

The starting ledger state consists of the default genesis objects, complemented by the working_bench package, a working_bench::tools::Simple object to use as the collection of dynamic fields, plus three gas coin transferred to the caller of the transactions.

The transactions that are executed are the following:

1. add_fields_tx: The transaction to call the package function tools::add_fields, sent and signed by the caller.
2. read_fields_tx: The transaction to call the package function tools::read_fields, sent and signed by the caller.
3. tools::remove_fields, sent and signed by the caller.

The complete scenario and partial scenarios are visualized in Figure G.2.

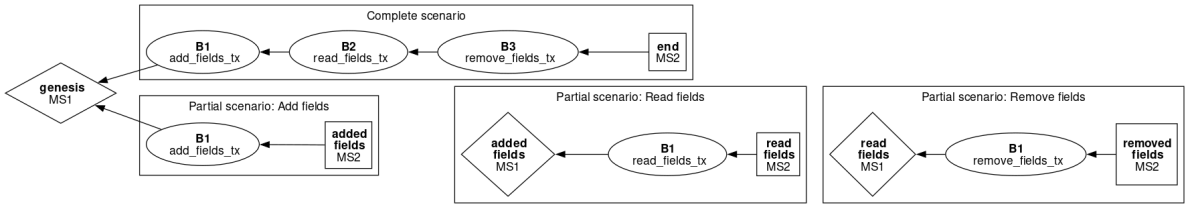


Figure G.2: Tangle representation of the dynamic_fields scenario.

The expected ledger end states are as follows:

Partial scenario: Add fields

- **Written:** We expect six new objects
  - The mutation of the Simple object acting as a collection.

  - The mutation of the gas coin owned by the caller.
  - The three fields added to the Simple object.
- **Deleted:** We expect two objects
  - The outdated version of the Simple object acting as a collection.
  - The outdated version of the gas coin owned by the caller.

Partial scenario: Read fields

- **Written:** We expect two new objects
  - The mutation of the Simple object acting as a collection.
  - The mutation of the gas coin owned by the caller.
- **Deleted:** We expect two objects
  - The outdated version of the Simple object acting as a collection.
  - The outdated version of the gas coin owned by the caller.

Partial scenario: Remove fields

- **Written:** We expect two new objects
  - The mutation of the Simple object acting as a collection.
  - The mutation of the gas coin owned by the caller.
- **Deleted:** We expect five objects
  - The outdated version of the Simple object acting as a collection.
  - The outdated version of the gas coin owned by the caller.
  - The three removed dynamic fields from the Simple object.

Complete scenario

- **Written:** We expect nine new objects
  - The three mutations of the Simple object acting as a collection. One mutation for every transaction.
  - The three mutations of each gas coin owned by the caller. One mutation for every transaction.
  - The three fields added to the Simple object.
- **Deleted:** We expect nine objects
  - The three outdated versions of the Simple object acting as a collection, following the mutation caused by each transaction.
  - The three outdated versions of each gas coin owned by the caller, following the mutations caused by each transaction.
  - The three removed dynamic fields from the Simple object during remove_fields_tx.

G.5. Dynamic Object Fields

This scenario adds three dynamic objects fields, and then reads and removes them through three successive transactions that rely on the working_bench package.

In addition we generate three partial scenarios for each gradual state transition. That is, from genesis to the added fields, from the added fields to the read fields, and from the read fields to the removed fields.

The starting ledger state consists of the default genesis objects, complemented by the working_bench package, a working_bench::tools::Simple object to use as the collection of dynamic fields, three more

working_bench::tools::Simple objects to add as dynamic fields, plus three gas coin transferred to the caller of the transactions.

The transactions executed are the following:

1. add_object_fields_tx: The transaction to call the package function tools::add_object_fields, sent and signed by the caller.
2. read_object_fields_tx: The transaction to call the package function tools::read_object_fields, sent and signed by the caller.
3. remove_object_fields_tx: The transaction to call the package function tools::remove_object_fields, sent and signed by the caller.

The complete scenario and partial scenarios are visualized in Figure G.3.

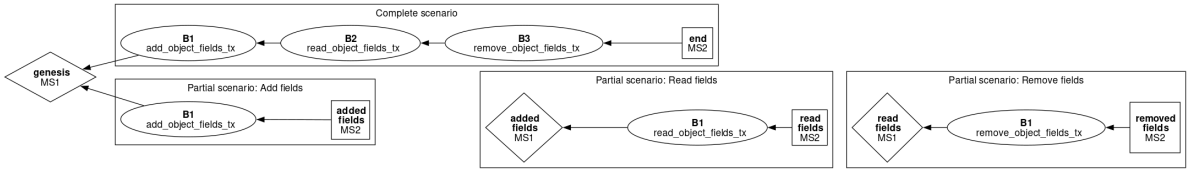


Figure G.3: Tangle representation of the dynamic_object_fields scenario.

The expected ledger end states are as follows:

G.6. Partial scenario: Add fields

- **Written:** We expect eight new objects
  - The mutation of the Simple object acting as a collection.
  - The mutation of the gas coin owned by the caller.
  - The three Field objects added to the Simple object acting as a collection.
  - The three mutated Simple objects that were added as dynamic object fields through the Field objects.
- **Deleted:** We expect five objects
  - The outdated version of the Simple object acting as a collection.
  - The outdated version of the gas coin owned by the caller.
  - The three outdated Simple objects added as fields.

G.7. Partial scenario: Read fields

- **Written:** We expect two new objects
  - The mutation of the Simple object acting as a collection.
  - The mutation of the gas coin owned by the caller.
- **Deleted:** We expect two objects
  - The outdated version of the Simple object acting as a collection.
  - The outdated version of the gas coin owned by the caller.

G.8. Partial scenario: Remove fields

- **Written:** We expect five new objects
  - The mutation of the Simple object acting as a collection.
  - The mutation of the gas coin owned by the caller.

  - The three mutated Simple objects that were removed from the collection.
- **Deleted:** We expect eight objects
  - The outdated version of the Simple object acting as a collection.
  - The outdated version of the gas coin owned by the caller.
  - The three removed Field objects from the Simple object acting as a collection.
  - The three outdated Simple objects that were removed from the collection and transferred to an address.

G.9. Complete scenario

- **Written:** We expect fifteen new objects
  - The three mutations of the Simple object acting as a collection. One mutation for every transaction.
  - The three mutations of each gas coin owned by the caller. One mutation for every transaction.
  - The three Field objects added to the Simple object.
  - Six more Simple objects accounting for the mutations while added and then removed via the Field objects.
- **Deleted:** We expect fifteen objects
  - The three outdated versions of the Simple object acting as a collection, following the mutation caused by each transaction.
  - The three outdated versions of each gas coin owned by the caller, following the mutations caused by each transaction.
  - The three removed Field objects from the Simple object during remove_object_fields_tx.
  - Six more Simple objects outdated while added and then removed via the Field objects.

G.10. Wrap Object

This scenario creates two single owned objects already present in the starting ledger state and wraps one of them into the other.

The transaction calls a package also present in the starting ledger state.

The starting ledger state consists of the default genesis objects, complemented by the working_bench package, a gas coin transferred to the caller of the transaction, a working_bench::tools::Simple and a working_bench::tools::SimpleWrapper object.

The transaction that is executed is the following:

- call_tx: The transaction to call the package function tools::wrap_simple, sent and signed by the caller with a working_bench::tools::Simple and a working_bench::tools::SimpleWrapper object as argument.

Tangle representation of the scenario is shown in Listing G.1:

Listing G.1: Tangle Representation of wrap$_{objectscenario}$

```
1     <genesis> (SEP) <- [MS1] <- call_tx(wrap object) <- [MS2]
```

The expected ledger end states are as follows:

- **written:** We expect two (2) new objects
  - The mutated gas coin owned by the caller.
  - The mutated `working_bench::tools::SimpleWrapper` object filled in `call_tx`.
- **deleted:** We expect three (3) objects

  - The outdated gas coin used in `call_tx`.
  - The `working_bench::tools::Simple` object wrapped in `call_tx`.
  - The outdated `working_bench::tools::SimpleWrapper` object filled in `call_tx`.

G.11. Unwrap Object

This scenario unwraps a working_bench::tools::Simple object from a working_bench::tools::SimpleWrapper object. This filled working_bench::tools::SimpleWrapper object is already present in the starting ledger state.

The transaction calls a package also present in the starting ledger state.

The starting ledger state consists of the default genesis objects, complemented by the working_bench package, a gas coin transferred to the caller of the transaction, a working_bench::tools::SimpleWrapper object wrapping a working_bench::tools::Simple object.

The transaction that is executed is the following:

- call_tx: The transaction to call the package function tools::unwrap_simple, sent and signed by the caller with a working_bench::tools::SimpleWrapper object as argument.

The tangle representation of this simple scenario is the same as the one in Figure G.1.

The expected end state of the ledger is:

- **written:** We expect three new objects
  - The mutated gas coin owned by the caller.
  - The mutated, empty `working_bench::tools::SimpleWrapper` object.
  - The created `working_bench::tools::Simple` object.
- **deleted:** We expect two deleted objects
  - The outdated gas coin used in `call_tx`.
  - The outdated `working_bench::tools::SimpleWrapper` object filled in `call_tx`.

G.12. Unwrap and Delete Object

This scenario unwraps and deletes a working_bench::tools::Simple object from a working_bench::tools::SimpleWrapper object. The filled working_bench::tools::SimpleWrapper object is already present in the starting ledger state.

The transaction calls a package also present in the starting ledger state.

The starting ledger state consists of the default genesis objects, complemented by the working$_{p}ackage, agascointransfer$ $tools :: SimpleWrapperobjectwrappingaworking\_bench :: tools :: Simpleobject.$

The transaction that is executed is the following:

- call_tx: The transaction to call the package function tools::unwrap_and_delete_simple, sent and signed by the caller with a working_bench::tools::SimpleWrapper object as argument.

Likewise, the Tangle representation of this scenario is the same as the previous scenario: Figure G.1.

The expected end state of the ledger is:

- **written:** We expect two new objects
  - The mutated gas coin owned by the caller.
  - The mutated, empty `working_bench::tools::SimpleWrapper` object.
- **deleted:** We expect two objects
  - The outdated gas coin used in `call_tx`.
  - The outdated `working_bench::tools::SimpleWrapper` object filled in `call_tx`.

G.13. Abort

This scenario aborts the execution of a transaction due to insufficient gas. The only object that is written is the mutated gas coin owned by the caller.

The transaction calls a package already present in the starting ledger state.

The starting ledger state consists of the default genesis objects, complemented by the working_bench package, and a gas coin transferred to the caller of the transaction.

The executed call transaction is the following:

- call_tx: The transaction to call the package function tools::create_simple, sent and signed by the caller.

To test the execution abort, the caller specifies a gas budget of 110 (minimum gas budget for the transaction to be valid), which is not sufficient to fulfill the gas requirements (295) of the tools::create_simple function.

This scenario is similar with previous simple scenarios, as it only consists of one call transaction. Therefore, the Tangle representation is similar to the one in G.1.

The end ledger state is:

- **written:** We expect one new object
  - The mutated gas coin owned by the caller.
- **deleted:** We expect one outdated object
  - The gas coin used in `call_tx`.

G.14. Shared Object

In this scenario, the Whiteflag algorithm is tested with multiple transactions that attempt to mutate a shared object. This scenario makes use of the DonutBox, a shared object introduced in the working_bench::donuts module. A DonutBox can contain a fixed number of Donuts. A user can try to get a Donut from the DonutBox by calling the function donuts::get_donut function if there are any Donuts left. The scenario is intended to create a competing situation in which two users simultaneously attempt to retrieve the last Donut from the DonutBox and only one of them can be successful.

This scenario defines 4 different users, each with their own address and gas coin. Furthermore, we define the DonutBox with a capacity of 3 donuts, leading to a competition for the last donut.

This scenario leverages the Whiteflag algorithm to determine a sequence in which the conflicting transactions are resolved.

The starting ledger state consists of the default genesis objects, complemented by the working_bench package with its modules working_bench::tools and working_bench::donuts, and a gas coin transferred to 4 unique users, owning their own address and gas coin.

The transactions executed are the following:

1. call_user1: The transaction to call the package function donuts::get_donut, sent and signed by the user 1.
2. call_user2: The transaction to call the package function donuts::get_donut, sent and signed by the user 2.
3. call_user3: The transaction to call the package function donuts::get_donut, sent and signed by the user 3.
4. call_user4: The transaction to call the package function donuts::get_donut, sent and signed by the user 4.

The scenario requires following Tagged Data Blocks which help build up the Tangle structure.

- tagged_data_block1

- tagged_data_block2
- tagged_data_block3

According to the protocol, the parents of a block must be defined in alphabetical order. This leads to difficulties when creating a Tangle structure where we want to control the Whiteflag walk to test a specific scenario, as Whiteflag depends on the order of the parents in the block. Block IDs get derived by hashing the block data, and therefore we cannot simply adjust these as desired, else the block would not match the given BlockID and we could run into protocol issues. To solve this problem, we use so-called ”proxy blocks”. Proxy blocks are Tagged Data Blocks that contain specific content which enables us to control the resulting Block IDs and therefore help us influence the Whiteflag walk.

In this scenario, we define following Proxy Blocks:

- proxy_user1: Tagged Data Block to influence the walk for the user 1.
- proxy_user3: Tagged Data Block to influence the walk for the user 3.
- proxy_user4: Tagged Data Block to influence the walk for the user 4.

Two sub scenarios are defined in which call_user3 and ‘call_user4 are issued simultaneously by different users. The only difference is where they attach, and the winners shall be different.

For the scenarios to be as realistic as possible, the Tangle structure consists of three Milestones, where:

- MS1 is loaded with the genesis objects
- MS2 references data transactions
- MS3 references the actual scenario transactions (call_user1, call_user2, call_user3, call_user4) and the tagged data blocks

The shared_object_user3_wins scenario is visualized in Figure G.4.

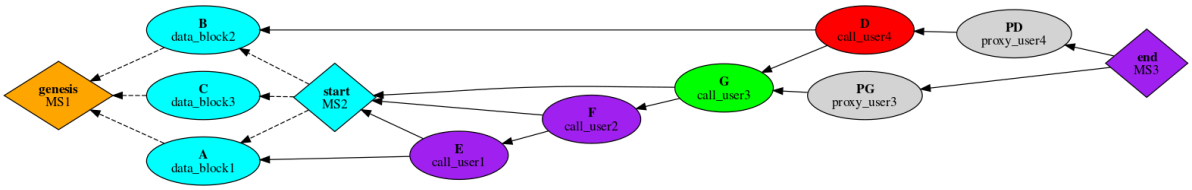


Figure G.4: Tangle representation of the shared_object_user3_wins scenario.

While reading the DAG structure(without considering the WhiteFlag algorithm) it is clear that `call_user1 < call_user2 < call_user3`.

It´s not clear if `call_user3 < call_user4 or call_user4 < call_user3`. This is where we need the WhiteFlag algorithm to determine the order.

Applying the WhiteFlag algorithm at MS3 gives us:

1. Start at MS3.
2. Visit the parents of MS3: PG and PD.
3. For PG:
    - Visit its parents: F
    - For F, visit its unvisited parent: E.
    - E’s parents are A and MS2, both of which are not part of the ordering.
4. For PD:
    - Visit its parents: D
    - For H, visit its parents: E and F, but both are already visited.
    - For D, visit its parent: E, but E is already visited.

the resulting order is: `E, F, G, D`.

The end ledger state is:

- written:
  - The mutated gas coin owned by the caller 1.
  - The mutated gas coin owned by the caller 2.
  - The mutated gas coin owned by the caller 3.
  - The mutated gas coin owned by the caller 4.
  - The DonutBox that was last touched by caller 4.
  - The Donut unwrapped by caller 1.
  - The Donut unwrapped by caller 2.
  - The Donut unwrapped by caller 3.
- deleted:
  - The gas coin used in call_user1.
  - The gas coin used in call_user2.
  - The gas coin used in call_user3.
  - The gas coin used in call_user3.
  - The DonutBox that was touched by call_user1.
  - The DonutBox that was touched by call_user2.
  - The DonutBox that was touched by call_user3.
  - The DonutBox that was touched by call_user4.

The shared_object_user4_wins scenario is visualized in Figure G.5.

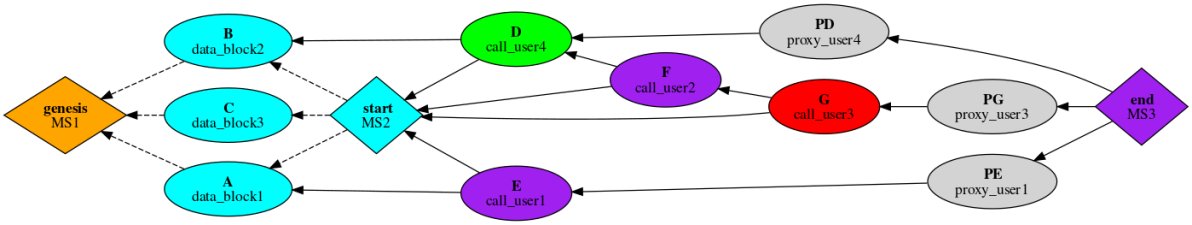


Figure G.5: Tangle representation of the shared_object_user4_wins scenario.

While reading the DAG structure (without considering the WhiteFlag algorithm) it is clear that `call_user4 < call_user2 < call_user3`.

It is not clear if `call_user1 < call_user4 or call_user4 < call_user1`. This is where we need the WhiteFlag algorithm to determine the order.

Applying the WhiteFlag algorithm at MS3 gives:

1. Start at MS3.
2. Visit the parents of MS3: PD, PE and PG.
   - For PD:
     - Visit its parents: D
     - D's parents are B and MS2, both of which are not part of the ordering.
   - For PE:
     - Visit its parents: E.
   - For PG:

  - Visit its parents: G.
  - For G, visit its parent: F
  - For F, visit its parent D, but D is already visited.
  - F's parent is MS2, which is not part of the ordering.

The resulting order is: `D, E, F, G`.

By attaching the transaction in another place, user 4 even makes it to the top of the order and will get the first donut.

The end ledger state is:

- written:
  - The mutated gas coin owned by the caller 1.
  - The mutated gas coin owned by the caller 2.
  - The mutated gas coin owned by the caller 3.
  - The mutated gas coin owned by the caller 4.
  - The DonutBox that was last touched by caller 3.
  - The Donut unwrapped by caller 1.
  - The Donut unwrapped by caller 2.
  - The Donut unwrapped by caller 4.
- deleted:
  - The gas coin used in call_user1.
  - The gas coin used in call_user2.
  - The gas coin used in call_user3.
  - The gas coin used in call_user4.
  - The DonutBox that was touched by call_user1.
  - The DonutBox that was touched by call_user2.
  - The DonutBox that was touched by call_user3.
  - The DonutBox that was touched by call_user4.