University of Padua

Department of Mathematics "Tullio Levi-Civita"

Master degree in Computer Science

Analysis of the implementation of the Stipula legal calculus using Distributed Ledger Technologies

Master's degree thesis

Supervisor

Professor Silvia Crafa

Graduand

Federico Zanardo

Academic year 2022-2023

Abstract

The purpose of this thesis is to provide a possible implementation of the domain-specific language *Stipula*. The purpose of this language is to assist professionals such as lawyers in programming *legal contracts*. The language is based on a set of programming abstractions that correspond to the distinctive elements that make up a legal contract, namely, permissions, prohibitions, obligations, asset transfer, and openness to external context, and that is likely to be executed on both centralized and distributed systems. The unique characteristics of the language have strongly driven the implementation of a particular architecture, which is a solution obtained from a combination of different approaches present in the current blockchain context.
In parallel with the development of a contract execution system, the aim of the project is also to provide secure mechanisms for *asset transfer*, to ensure that the transferred sum is not altered during the sending and receiving of a transaction, and avoid attacks such as *double-spending*.
The implementation illustrated in this thesis is the first version that provides the general idea of the project, oriented towards the execution in different types of distributed systems, ranging from a simple client-server system to a network of replicated partially trusted nodes. To illustrate the implementation, the thesis fully discusses the case of non-trivial contracts such as the trading of assets and the rent of a bike. The final part of the paper presents the missing functionalities, addresses the current limits of the version presented, possible solutions proposed, and possible future developments of the project are introduced. In particular, for this last point, future developments are understood as evolutions from the point of view of security, usability, computation, and efficiency for the execution of legal contracts in distributed systems.

Ringraziamenti

Desidero ringraziare la Professoressa Silvia Crafa per avermi dato la possibilità di svolgere il progetto di tesi a partire da un Suo lavoro di ricerca e di avermi dato ampie libertà riguardo le scelte di progettazione e di implementazione.

Desidero ringraziare Enrico che mi ha dato un forte sostegno in uno dei momenti più complicati. Non potrei chiedere un fratello migliore.

Desidero ringraziare i miei genitori e tutti i miei cari per esserci sempre stati nei momenti più importanti e difficili.

Ringrazio tutti gli amici che mi sono stati vicino e mi hanno sostenuto. In particolare, vorrei ringraziare Stefano, Lorenzo, Massimo, Manuele, Nicole, Francesco e Mariagrazia che mi sono stati molto vicini in questi ultimi ostici mesi.

Ringrazio Alessia per avermi dato un grande sostegno e per aver vissuto dei momenti meravigliosi.

Grazie di cuore a tutti per rendere la mia vita piena di meravigliosi ricordi.

Padova, 21 April 2023 Federico Zanardo

Acknowledgements

I would like to thank Professor Silvia Crafa for giving me the opportunity to carry out the thesis project starting from her research work and for giving me ample freedom regarding design and implementation choices.

I would like to thank Enrico who gave me strong support in one of the most complicated moments. I could not ask for a better brother.

I want to thank my parents and all my loved ones for always being there in the most important and difficult moments.

I thank all the friends who have been close to me and supported me. In particular, I would like to thank Stefano, Lorenzo, Massimo, Manuele, Nicole, Francesco and Mariagrazia who have been very close to me in these last difficult months.

I thank Alessia for giving me great support and for having lived such wonderful moments.

Heartfelt thanks to everyone for making my life full of wonderful memories.

Padova, 21 April 2023 Federico Zanardo

Contents

1 Introduction **1**
- 1.1 Context . . . 1
- 1.2 Aim of the thesis . . . 1
- 1.3 Personal motivations . . . 5
- 1.4 Document structure . . . 6

2 Stipula **7**
- 2.1 Context . . . 7
- 2.2 Legal calculi . . . 8
- 2.3 Code-driven normativity . . . 8
- 2.4 Building blocks of Stipula . . . 9
  - 2.4.1 Examples of contracts in Stipula . . . 9
  - 2.4.2 Agreement . . . 11
  - 2.4.3 Permissions and prohibitions . . . 12
  - 2.4.4 Assets . . . 12
  - 2.4.5 Obligations . . . 13
  - 2.4.6 Third party enforcements . . . 13

3 Analysis and design of Stipula platform **15**
- 3.1 Blockchain layers . . . 15
- 3.2 Basic ideas . . . 16
- 3.3 Architecture . . . 19
  - 3.3.1 Message Service . . . 20
  - 3.3.2 Compiler . . . 20
  - 3.3.3 Virtual Machine . . . 21
  - 3.3.4 Consensus . . . 22
  - 3.3.5 Storage . . . 22
  - 3.3.6 Commitment . . . 23
  - 3.3.7 Communication protocols . . . 23
- 3.4 Interaction between modules . . . 23
- 3.5 Asset management . . . 24
  - 3.5.1 Definition of assets . . . 24
  - 3.5.2 Transfer of assets . . . 25

4 Implementation **31**
- 4.1 Introduction to basic concepts . . . 31
  - 4.1.1 Contracts and contract instances . . . 31
  - 4.1.2 Asset . . . 33

4.2 Libraries . . . 34
4.2.1 Crypto . . . 34
4.2.2 Data structures . . . 34
4.3 Message Service . . . 35
4.3.1 MessageServer . . . 35
4.3.2 ClientHandler . . . 35
4.3.3 ClientConnection . . . 35
4.3.4 Messages . . . 35
4.3.5 Interaction with Storage . . . 38
4.4 Compiler . . . 39
4.4.1 Grammar, lexer e parser . . . 40
4.4.2 Generation of the bytecode . . . 42
4.5 Stipula bytecode . . . 42
4.5.1 Types . . . 44
4.5.2 Instructions of the bytecode language . . . 45
4.5.3 Function types . . . 49
4.6 Virtual Machine . . . 51
4.6.1 Requests queue . . . 51
4.6.2 Legal Contract Virtual Machine . . . 53
4.6.3 Script Virtual Machine . . . 54
4.6.4 Description of the execution flow of a function of a contract . . . 58
4.6.5 Pay-to-Party . . . 60
4.6.6 Obligations . . . 60
4.7 Storage . . . 61
4.7.1 LevelDB . . . 61
4.7.2 Structure . . . 63
4.8 Examples . . . 65
4.8.1 Asset swap . . . 65
4.8.2 Asset swap with scheduled event . . . 79
4.8.3 Bike rental . . . 83
4.9 Project management . . . 92
4.9.1 Pipeline . . . 92
4.9.2 Issues, milestones ans releases . . . 92
4.9.3 Installation . . . 93

5 Missing features and future developments 97
5.1 Missing features . . . 97
5.1.1 Language features not implemented in the current version . . . 97
5.1.2 Single-use-seals merge . . . 98
5.1.3 Creation of assets and their distribution . . . 103
5.2 Optimizations . . . 104
5.3 Limits of the architecture . . . 105
5.3.1 Computational and memory resources required . . . 105
5.4 Current version security issues . . . 106
5.5 Future improvements . . . 107
5.5.1 Implementation of the consensus module and communication protocols . . . 107
5.5.2 Implementation of the commitment module . . . 108
5.5.3 Fees for performance of a contract . . . 108
5.5.4 Script language extension . . . 108

5.5.5 Implementation of additional software 109

6 Conclusion **111**
6.1 Design considerations 111
6.1.1 Virtual Machine and Stipula bytecode 111
6.1.2 Asset Management and Script Language 111
6.1.3 Distributed context and consent 112
6.2 Implementation consideration 112
6.2.1 Structure of the project 113

A Examples of contracts and execution of contracts **115**
A.1 Asset swap 115
A.1.1 Complete code 115
A.2 Asset swap with scheduled event 117
A.2.1 Complete code 117
A.2.2 Complete example of execution 119
A.3 Bike rental 129
A.3.1 Complete code 129
A.3.2 Complete example of execution 131

B Grammar **145**
B.1 Lexer rules 145
B.2 Parser rules 146

C Pipelines **149**
C.1 run-tests.yml 149
C.2 create-and-push-docker-image.yml 150

D Gradle **151**
D.1 build.gradle 151

E Docker **153**
E.1 Dockerfile 153
E.2 docker-compose.yml 154

Bibliography **155**

Sitography **157**

List of Figures

3.1 Possible configurations of the *Stipula* architecture. 18
3.2 Complete architecture of all modules and interactions between them. . 19

4.1 Current state of the implemented architecture. 32
4.2 Virtual machine. 52
4.3 The instruction sets of the two virtual machines. 55
4.4 Sets of the types of the two virtual machines. 56
4.5 Flow of the execution of a function of a function of a contract. 59
4.6 Flow of execution of an obligation. 62
4.7 Structure of the *Storage* module. 63
4.8 New contract upload execution flow. 70
4.9 Execution flow for Alice's funds read request. 71
4.10 Various project releases. For each release it is possible to download the code and start an instance of *Stipula*. 93
4.11 Milestones completed. 94
4.12 Milestone opened. 94

5.1 Example of single-use-seals merge. 99

List of Tables

2.1 Correspondence between legal elements and *Stipula* features (Silvia Crafa, 2022). 9

4.1 Table of *Stipula bytecode* instructions. 43
4.2 Table of *Script Virtual Machine* instructions. 54

Chapter 1

Introduction

This chapter introduces the general problem of the thesis, the technologies used for the development of the proposed solution and the structure of the document.

1.1 Context

The digital revolution is having a significant impact on the *legal domain*, and one of the main challenges in dealing with legal documents is their complexity. *Legal contracts*, which give rise to a binding legal relationship, are a specific subset of legal documents that are the focus of this research. The principle of *freedom of form* in contractual law allows parties to express their agreement using any language or medium, including programming languages. This creates a need for high-level programming languages that are *easy for non-experts to understand*, as genuine agreement can only be reached if the parties are aware of the computational effects of their code. The researchers propose a new domain-specific language called **Stipula** (Silvia Crafa, Cosimo Laneve and Giovanni Sartor, 2022 and Silvia Crafa, Cosimo Laneve and Giovanni Sartor, 2021), which uses concise and intelligible primitives that have precise correspondence with the distinctive elements of legal contracts. *Stipula* is based on the theory of concurrency in computer science, which provides a rich toolset of formal techniques to verify the properties and correctness of software contracts.

1.2 Aim of the thesis

The language definition of *Stipula* is *implementation-agnostic* and can be either implemented as a centralized platform or run on top of a distributed system like a blockchain. Building a distributed system for managing and executing legal contracts involves the design and development of several very complex components. The *Stipula* language guarantees non-trivial primitives for the execution of a contract, such as the *automatic execution* of some parts of a contract and the secure transfer of assets between actors and contracts. The *scheduling* of an automatic execution of some parts of a contract when certain conditions are met is not a feature offered by other languages. For the legal context, however, this represents an important feature for the implementation of compliance with some *obligations* of a contract, which, if not respected, trigger *penalties* for one or more parties to the contract same. The *secure transfer* of assets is a subject that is treated in different languages and there are different approaches to guarantee

the fundamental properties for the transfer of assets, such as avoiding the possibility of *double-spending*, avoiding the generation of a quantity or lose an amount of assets during a transfer. Furthermore, different blockchains use different representations for the implementation and management of assets within their network and this is certainly one of the most delicate and important aspects to take into consideration when designing a language that allows writing contracts with value legal.

The idea of the *Stipula* language comes from the languages present in the blockchain context, proposing new primitives useful for creating legal contracts. However, a distributed system for executing *Stipula* contracts can in turn be based on another distributed system, such as a blockchain. The idea behind the implementation of the *Stipula* language is to *separate* the contract execution layer from the information storage layer. In doing so, the blockchain is used as a secure ledger in which to store the information produced by the upper layer that is in charge of executing *Stipula* contracts. This separation allows you to leave out a whole series of issues that are delegated to the lower layer (such as the immutable archiving of information), and to concentrate on providing functions dedicated to the drafting and execution of legal contracts.

As with blockchains like Ethereum, the execution of a smart contract requires the *consent* of the network in order to validate the result obtained. Consensus is one of the most complex aspects of distributed systems, as it requires the nodes of a network to agree on a particular result of a computation. This theme involves different aspects such as the communication of the nodes. In fact, there must be protocols that allow the nodes to communicate effectively and efficiently, both information necessary for maintaining the network (i.e., node discovery) and the information to be submitted for consent (the results obtained from the execution of a smart contract). In particular, achieving consensus in a distributed system is a complex task as it involves several considerations. Firstly, the system must be able to *tolerate* node errors, message losses, network partitions, and other failures while ensuring consistency and correctness (*fault tolerance*). Secondly, *scalability* is crucial as the number of nodes in the system increases. The consensus protocol must adapt to the growing number of nodes to avoid bottlenecks and delays. Finally, *security* is also paramount as the consensus protocol should prevent malicious actors from compromising the system. Attackers may try to manipulate messages, disrupt communication, or impersonate other nodes to compromise the integrity of the system.

Furthermore, in a distributed context, aspects such as performance and computational resources are very important requirements to take into consideration. Requiring the execution of nodes with high computational and memory capacities would not help distribute the network as the management and maintenance costs of these nodes can only be supported by a limited number of entities. Therefore, the implementation must take into account:

1. Store as little information as possible, ensuring the correct execution of contracts and asset transfers;

2. Minimize the computational resources required: an implementation that can also be performed on devices with a low computational capacity allows to increase the distribution of the network. However, this point collides with the peculiar features required by the language. Therefore, compromises need to be found.

As far as computation is concerned, a point that can help to minimize the demand for computational resources is to opt for a *compilation* of the language instead of

carrying out a pure *interpretation*. Compiling the language will certainly take more time when you upload a contract to the network. However, when the contract has to be performed, the original contract will not be performed, but its compiled will be performed, obtaining an increase in performance compared to performing the interpretation of the original contract each time.

The thesis work took into consideration both the aspects strictly related to language and the aspects related to distribution. Indeed, the design phase took into account the need for specific components for the implementation of a distributed system. The realized architecture implements all the most important features of the *Stipula* language, such as:

1. Management of the agreement between the parties to start a new contract
2. Management of the progress of a contract
3. The secure transfer of assets between actors and contracts
4. The management of the execution of specific parts of the contract for the implementation of penalties, if the conditions arise
5. Management of judicial enforcement and exceptional behavior

The most interesting and most complex parts of the architecture involve the **compiler**, the **virtual machine** and the development of a **bytecode language** and **asset management**.

Precisely to try to obtain better performance in the execution phase of the contracts, the design required the development of a compiler, which could translate the contract in *Stipula* into a target language suitable for execution on a protected environment (a virtual machine). Therefore, a language called **Stipula bytecode** was designed, and represents the output produced by the compiler. By doing so, it is possible to carry out the compilation phase once only when the contract is loaded into the distributed platform. Every time you want to execute a contract, a specific virtual machine will execute the compile written in *Stipula bytecode*. Compiler development tries to do as many static checks as can be done. Since the *Stipula* language is an **untyped** language, the compiler also performs *type inference*. This task is not always successful and it may happen that in certain situations the compiler is not able to determine the type of a variable. Therefore, a syntax was developed in *Stipula bytecode* that allows you to notify the virtual machine to perform a *runtime type check*.

The *Stipula bytecode language* is a language designed to run on a **stack-based virtual machine**, that is, the variables of the program are loaded on a stack and the *Stipula bytecode* language operates on that stack by removing or adding elements. The *Stipula bytecode* language has the same expressiveness as the *Stipula* language. The advantage of using this bytecode language is the ease of executing a contract, as a program written in bytecode directly contains the instructions that allow it to operate on the stack and do not require a further interpretation step. Unlike the *Stipula* language, the bytecode language allows **types** for variables and also takes into account the presence of variables without a specific type, precisely because of the untyped behavior of the *Stipula* language.

The *virtual machine* represents one of the most interesting and complex modules of the whole architecture. This component takes care of:

1. Execute contracts written in Stipula bytecode;

2. Manage asset transfers between users and contracts;
3. Manage the scheduling of events for the automatic execution of certain obligations;
4. Manage the opening of new contracts and the progress of the same.

The first activity that the virtual machine has to manage is the opening of a contract which corresponds from a legal point of view to the *agreement of minds* between two or more actors. Therefore, it is necessary for this component to correctly manage the communication between the actors to make sure that everyone is in agreement on opening a new contract. The main job of the virtual machine is to execute a contract and ensure that the execution is done in a precise state of the contract, as well as ensuring that the caller matches the correct participant of the contract and checks on the input parameters for the execution of the contract. In addition to this, the virtual machine manages all the event *scheduling* mechanism for the automatic execution of *obligations*: this process starts from the execution of a contract which invoked the creation of a *penalty*, in case a participant does not respect the agreements stipulated at the opening of the contract, and proceeds with the control of certain conditions at a precise moment. If the conditions for applying the penalty exist, the virtual machine will execute a specific portion of code that encodes the obligation, otherwise the virtual machine will avoid applying the penalty. Another very important aspect of the virtual machine is the management of **asset transfers**. In fact, all asset transfers to a contract are verified by the virtual machine, which ensures:

1. To receive the right amount of assets established by the contract;
2. To check that during the transfer of a certain amount of assets, no amount of assets is generated or lost in the void;
3. To check that the funds received actually belong to the rightful owner;
4. To check that the user who wants to make a payment is not attempting to carry out a **double-spending** attack on the system.

Conversely, when a contract makes a payment to a user, the machine must ensure that an unexpected amount of assets is not lost or generated out of thin air during the transfer.

Asset management is one of the most delicate aspects of architecture. Asset transfers directly between users and other users are not permitted, but all transfers must be by way of a contract. In the thesis an example will be illustrated which will illustrate how two actors can exchange two assets by means of a contract (4.8.1). The reason is that this would have defeated the purpose of *Stipula* for certain applications, so a transfer can be:

1. *Pay-to-Contract*: it is a transfer that takes place from a participant of the contract to a contract;
2. *Pay-to-Party*: it is a transfer that takes place from a contract to a participant in the contract.

In order to be able to transfer assets, it is necessary to decide which **asset representation model** to adopt. Currently, there are two models in the blockchain context: the **account-balance-based model** and the **UTXO model** (*Unspent Transaction Output*). The account-balance-based model, used for example by Ethereum, is the

simplest model from a conceptual and implementation point of view. In this model, each user (represented by an alphanumeric address) is associated with the *balance* of a specific asset. Each time a transaction is made, the balance of the asset is updated. This model has severe limitations from the point of view of *performance*, as it does not allow the parallel execution of several transactions, and from the point of view of *privacy*, as in this model it is very easy to trace the funds associated with the users. The model used for the implemented architecture has been defined **single-use-seal**, and is a simplified version of the UTXO model used by Bitcoin. This model is more complex from a conceptual and implementation point of view but allows to obtain performance and privacy advantages, as well as *partially mitigating the double-spending* attack by its nature. A single-use-seal can be seen as a box that contains a certain amount of assets and is sealed by a single-use seal, which can only be broken by the owner of that amount of assets. To execute a transaction against a contract, the owner must break the seal, in order to allow the contract to withdraw the required funds. This seal can only be broken by the owner who is able to provide **cryptographic proof of ownership** of the asset amount. By doing so, a contract to withdraw funds directly from a user's wallet is denied, as happens in Ethereum. The cryptographic proof that allows to break the seal consists of a program written in a language called **Script** (Antonopoulos, 2017). In particular, a single-use-seal is **locked** by a program written in **Script**, defined as `lockScript`. The cryptographic proof that the user must provide to prove that he is the actual owner of the funds is a program written in Script, defined as `unlockScript`. The latter program is the one that allows you to unlock the `lockScript` script and consequently the funds contained in the seal. This *Script* language is a very similar language to the *Stipula bytecode* language and therefore programs written in *Script* can be executed in a virtual machine similar to the one for *Stipula bytecode*. Unlike the bytecode language, the *Script* language has a different and separate set of instructions from that of *Stipula bytecode*.

The seal is defined *single-use* because it can only be used once for an asset transfer, after which it is destroyed and can no longer be used by the user for other transfers. Instead, when a contract needs to make a *Pay-to-Party*, it creates new single-use-seals to be sent to one or more users.

1.3 Personal motivations

I attended several artificial intelligence courses during my master's degree. I was very fascinated by it, however I sensed that perhaps it is not the right career path for me. For the oral exam of the course *Advanced Topics in Programming Languages*, held by Professor Silvia Crafa, I had the opportunity to analyze distributed systems and in particular the blockchain field. I was introduced to the research project concerning the development of a programming language aimed at the creation of legal contracts: *Stipula*. I accepted the proposal to carry out a thesis work on this topic as it allowed me to analyze and deal with several separate topics, such as distributed systems, the analysis and implementation of a programming language, computation and cryptography. I was very enthusiastic about this project and I am passionate about studying distributed systems and finding solutions that are inspired by currently existing systems, but adapting them in a different key for this project.

1.4 Document structure

Illustrate the structure of the document:

The second chapter introduces the context of application of this language and all the main aspects of the language.

The third chapter describes the proposed solution for the language implementation, focusing on asset management and the components needed for the general architecture. Furthermore, the implementation is contextualized in a distributed system, and therefore specifying all the modules necessary to operate in a similar context.

The fourth chapter describes the design and implementation of the architecture. All the modules that make up the architecture and the interactions between them are explained. In addition, examples of contract execution will be illustrated.

The fifth chapter describes the missing features to have a complete implementation of the language. Furthermore, the limits of the implemented architecture are presented, the possible optimizations that can be applied to it to increase the performance and future developments of the project in its entirety.

The sixth chapter provides a brief summary emphasizing the complexity of the design, both for the implemented architecture and for future developments, and of the problems faced.

Chapter 2

Stipula

This chapter introduces all the fundamental concepts of the *Stipula* language. Its functionality and the main characteristics of the language are analyzed.

2.1 Context

Ethereum's introduction of smart contracts brought about the *Code is Law* principle (Lawrence Lessig, 1999), which relies on software code to provide a clear definition and automatic execution of transactions between parties who do not trust each other. In case of disputes, the code of the contract is publicly available and takes precedence. This principle is based on the idea that trust is built into transparent intermediary algorithms of the blockchain. As a result, governments recognize the legal value of smart contracts and programs operating over distributed ledgers.

Code-Driven Law (*The CoHuBiCoL research project* 2019) is a growing trend that uses software to represent or enact legislation or regulation. Technologies like Rules as Code (*Cracking the Code* 2020), Catala (*Catala in action* 2022), and Akoma Ntoso (*Akoma Ntoso* 2018) are used to create a machine-consumable version of some types of rules issued by governments and public administrations, such as tax offices, student grant provisions, or social security agencies. This helps to identify potential inconsistencies in regulations, reduce the complexity and ambiguity of legal texts, and support the automation of legal decisions. Instead of relying on ex-post enforcement by third parties like courts and police, the rules hardwired into code are enforced ex-ante, making it very difficult for people to breach them in the first place (Primavera De Filippi and Samer Hassan, 2016).

However, transposing legal rules into technical rules is a delicate process since the inherent ambiguity of the legal system is necessary to ensure a proper application of the law on a case-by-case basis. The process of translating parties' intentions, promises, actions, powers, and prohibitions into computer code is problematic and does not solve the problem but moves it into another dimension. Additionally, code-driven law is based on the automation of compliance with pre-set rules, leaving no room for disagreement about the right way to interpret the norms. This potentially reduces the capability of individual human beings to invoke legal remedies.

For example, the Code is Law principle of Ethereum declined with the famous TheDAO attack (David Siegel, 2016). From the Code is Law perspective, a problem in the source code leading to unexpected behavior of the smart contract is a feature of the code and not an error. However, the first hard fork of the Ethereum blockchain showed

that this principle is not practical when large sums of money are at stake. Furthermore, blockchain technology does not hardwire trust into algorithms but reassigns trust to a series of actors who implement, manage and enable the functioning of this technological platform.

2.2 Legal calculi

Legal contracts are defined as agreements that create a legally binding relationship or have legal effects. While parties are free to express their agreement using any language or medium, including a programming language, a contract only produces the intended effects if it is legally valid. This raises both legal and technological issues when it comes to software-based contracts.

Different kinds of software-based solutions can be valuable in the different phases of a legal contract's lifecycle, which goes through negotiation, storage/notarizing, performance, enforcement and monitoring, modification, and dispute resolution. Therefore, several projects are being developed for defining code-driven legal contracts. The main problem is defining a suitable programming language to write legal contracts, which should be easy-to-use and understand for legal practitioners while still being expressive and precise.

The *Stipula* programming language is proposed as a solution, which is an intermediate domain-specific language. The language's basic primitives are designed to map the building blocks of legal contracts into template programs and design patterns. The definition of *Stipula* is influenced by the theory of concurrent systems, and a legal contract is interpreted as an interaction protocol. Additionally, the language definition is *implementation-agnostic* and can be either implemented as a centralized platform or run on top of a distributed system like a blockchain. A prototype centralized implementation of *Stipula* as a Java application is available (Silvia Crafa, Cosimo Laneve and Adele Veschetti, 2022a).

While only a concrete implementation can address specific issues, studying the theory of a domain-specific legal calculus (2022 and Vimal Dwivedi, Vishwajeet Pattanaik, Vipin Deval, Abhishek Dixit, Alex Norta and Dirk Draheim, 2021) is a first step in shedding some light on the digitization of legal texts.

2.3 Code-driven normativity

A preliminary interdisciplinary study found that most actual legal contracts are composed of several basic elements (see table 2.1). These elements include the **meeting of the minds**, which refers to the agreement of the contract's parties to its terms, and marks the moment when legal effects take place. They also include **permissions**, **prohibitions**, and **obligation** clauses that may change dynamically, such as the right to use a product until a certain date. In particular, permissions correspond to the possibility of performing an action at a certain stage, prohibitions correspond to the interdiction of doing an action, while obligations are recast into commitments that are checked at a specific time limit and issue a corresponding penalty if the obligation has not been met. Other elements include the **transfer of assets or currency**, the possibility of **external conditions or data** affecting the contract, and the ability to **activate judicial enforcements** in the case of dispute resolution.

Table 2.1: Correspondence between legal elements and *Stipula* features (Silvia Crafa, 2022).

Legal contracts	Stipula contracts
Meeting of the minds	Agreement primitive
Permissions, prohibitions	State-aware programming
Obligations	Event primitive
Currency and tokens	Asset-aware programming
Openness to the environment	Intermediary pattern
Judicial enforcement and exceptional behaviours	Authority pattern

The *Stipula* language was designed to easily map these basic elements of legal contracts into template programs and design patterns. The **agreement** construct directly encodes the meeting of the minds, while normative elements such as permissions, prohibitions, and obligations are expressed using a **state-aware programming style** inspired by the state machine pattern used in smart contracts, such as *Solidity* (*Solidity Documentation: State Machine Common Pattern*) and *Obsidian* (*Obsidian: A safer blockchain programming language* 2018).

Asset manipulation is also syntactically distinguished from standard operations to emphasize that assets *cannot be destroyed or forged*, but *only transferred*. Assets are a specific value type and *Stipula* wants to promote an **asset-aware programming** (Franklin Schrans, Susan Eisenbach and Sophia Drossopoulou, 2018, A. Das, S. Balzer, J. Hoffmann, F. Pfenning and I. Santurkar, 2021, Sam Blackshear and et al., 2021). Clauses dependent on external data are implemented by an intermediary party responsible for retrieving data from an external source agreed upon in the contract.

Dispute resolution, judicial enforcement of legal clauses, and exceptional behaviors are also included in the contract by assigning legal responsibility to an intermediary party that interfaces with a court or an *Online Dispute Resolutions platform*, as *The European ODR platform*. This approach differs from relying on Oracles web services, which cannot be legally held accountable.

2.4 Building blocks of Stipula

It is worth to notice that a *Stipula* contract begins with the keyword `stipula` and define *assets* and *fields* that are used therein. We also observe that *Stipula* is **untyped**, to keep a simple syntax. However, a *type inference system* that allows one to derive types has been designed.

2.4.1 Examples of contracts in Stipula

This short section introduces two examples of contracts, written in *Stipula*, to illustrate the structure and main elements that make up a contract in *Stipula*. In the following sections, starting from the bike rental contract, all the fundamental blocks of a *Stipula* contract will be illustrated.

Asset swap

In this example, there are two actors, Alice and Bob, who want to trade two assets. For simplicity, the price variation that these assets may have over time is not taken

into consideration, the exchange rate of these two assets is fixed by the parties to the contract when a new instance of the contract is made.

The complete code of the contract written in *Stipula* is the following:

```
stipula SwapAsset {
  asset assetA:stipula_assetA_ed8i9wk,
  ↪  assetB:stipula_assetB_pl1n5cc
  field amountAssetA, amountAssetB
  init Inactive

  agreement (Alice, Bob)(amountAssetA, amountAssetB) {
      Alice, Bob: amountAssetA, amountAssetB
  } ==> @Inactive

  @Inactive Alice : depositAssetA()[y]
      (y == amountAssetA) {
          y -o assetA;
          _
  } ==> @Swap

  @Swap Bob : depositAssetBAndSwap()[y]
      (y == amountAssetB) {
          y -o assetB
          assetB -o Alice
          assetA -o Bob;
          _
  } ==> @End
}
```

Bike rental

The example that will be presented was taken from one of the papers describing the design of *Stipula* (Silvia Crafa, Cosimo Laneve and Giovanni Sartor, 2021, Silvia Crafa, Cosimo Laneve and Giovanni Sartor, 2022 and Silvia Crafa, 2022).There are two actors, one represents a company that rents bicycles for a defined amount of time and for a certain amount of money, and the other represents a customer of that company. When the customer wants to use the service offered by the company, both decide to create a new instance of the contract (*agreement* phase). The company will provide a certain code which will allow the user to unlock the bicycle. When the user has provided the amount of money needed to use the service, the user will receive the code from the company and the money will be deposited into the contract. Furthermore, the obligation will be set that if the user does not stop using the service offered by the company within a certain period of time, the money will be sent to the company and the user will be notified of the term of using the service.

The complete code of the contract written in *Stipula* is the following:

```
stipula BikeRental {
  asset wallet:stipula_coin_asd345
  field cost, rentingTime, use_code
  init Inactive

```

```
6       agreement (Lender, Borrower)(cost, rentingTime){
7           Lender, Borrower: cost, rentingTime
8       } ==> @Inactive
9
10      @Inactive Lender : offer(z)[] {
11          z -> use_code;
12          _
13      } ==> @Proposal
14
15      @Proposal Borrower : accept()[y]
16          (y == cost) {
17              y -o wallet;
18              use_code -> Borrower;
19              now + rentingTime >>
20                  @Using {
21                      "End_Reached" -> Borrower
22                      wallet -o Lender
23                  } ==> @End
24      } ==> @Using
25
26      @Using Borrower : end()[] {
27          wallet -o Lender;
28          _
29      } ==> @End
30  }
```

2.4.2 Agreement

One key aspect of a legal contract is the agreement between parties, where they come to a mutual understanding and give consent to the terms of the contract (*meeting of minds*). *Stipula* provides a specific primitive, called **agreement**, to indicate when parties have reached consensus on the contractual arrangement they wish to establish. As an example, consider a contract regulating a bike rental service. The following *Stipula* code implements the agreement phase:

```
6   agreement (Lender, Borrower)(cost, rentingTime) {
7     Lender, Borrower: rentingTime, cost
8   } ==> @Inactive
```

The code is meeting a `Lender` and a `Borrower` to agree on both the `rentingTime` and on its `cost`. After the agreement the contract starts and it goes into a state `@Inactive` that expresses that no rent will occur until the payment. The contract can also have a variation which involves an **authority** responsible for monitoring contextual constraints, such as obligations related to storage and care, or the proper use of goods. This `Authority` is also responsible for managing litigations and dispute resolution. In this case, the agreement function would be:

```
6   agreement (Lender, Borrower, Authority)(cost, rentingTime) {
7     Lender, Borrower: rentingTime, cost
8   } ==> @Inactive
```

This code express the fact that only the `Lender` and the `Borrower` agree on both `rentingTime` and `cost`, while the `Authority`, which also engage in the meeting of minds, is the pointer to a *third party* that will **supervise** `Lender` and `Borrower` behaviours.

2.4.3 Permissions and prohibitions

Legal contracts have a unique characteristic where the set of normative elements, such as **permissions** and **prohibitions**, often change based on actions taken or not taken. To account for these changes, *Stipula* adopts a *state-machine programming approach*, which is a widely supported pattern in programming languages using ad-hoc libraries or modules. For example, in a bike rental contract, once the `Lender` and `Borrower` agree on the rental period and cost, the `Lender` is not allowed to prevent the `Borrower` from paying for the service and using the bike. *Stipula* implements this functionality by enabling the contract to proactively monitor changes in the bike's status, such as storing a temporary access code to prevent the `Lender` from revoking the rental. The following code defines the function `offer` that can be invoked by `Lender` when the contract is in state `@Inactive` to send an access code to be used by the `Borrower`:

```
10    @Inactive Lender : offer(x) {
11      x -> code
12    } ==> @Payment
```

Naturally, the `Borrower` is not informed of the value of the `code` prior to payment for the service. In other words, the preceding excerpt authorizes the `Lender` to invoke `offer` function in the `@Inactive` state. If no further function is defined in `@Inactive`, the contract will prohibit other parties from taking any action at this stage. Once the code is received, the contract will move to the `@Payment` state, where presumably the `Borrower` will pay (in fact, the `Borrower` is allowed to pay) for the rental.

It's worth noting that the aforementioned code also emphasizes that the `Lender` trusts the contract to act as an intermediary that can store relevant information (such as assets). Indeed, `x -> code` stores the value sent by the `Lender` in a contract field called `code`, which cannot be accessed outside the contract.

2.4.4 Assets

Another key characteristic of legal contracts is the handling of **assets**, such as currency for *payments* and *escrows*, as well as tokens that can represent securities and provide digital ownership of physical goods. In the example of the bike rental, instead of using a simple numeric code, a more innovative approach could involve a unique token that grants access to the bike's smart lock. Traditionally, the `Borrower` pays the `Lender` with a credit card before using the bike, and the contract only specifies the transaction through a normative clause, with no guarantee of its occurrence (in case of dispute, one party has to go to court). However, *Stipula* enables digital legal contracts that automatically handle asset transfers, thus eliminating intermediaries even in payments. Moreover, *Stipula* allows legal contracts to temporarily retain assets and decide to redistribute them when certain conditions are met. Therefore, the language treats assets as first-class values and provides specific operations for their management. For example, the following function is defining the payment of the rental by `Borrower`, which sends an asset `y` (the argument is in square brackets) to the contract:

```
15    @Payment Borrower : accept[y]
16      (y == cost) {
17        y -o wallet
18        use_code -> Borrower
19    } ==> @Using
```

The function call has a precondition (line 2) that ensures the correctness of the fee paid by the `Borrower` before executing the operation at line 3, which transfers ownership of the asset `y` to the contract and stores it in the asset field `wallet`. By explicitly marking asset movements with the ad-hoc operator `-o` and separating it from `->`, the language promotes a safer programming discipline that *reduces the risk of double spending*, *accidental loss*, or *locked-in* assets. Notably, the contract does not immediately forward the payment to the `Lender`; instead, it retains the payment until the rental period terminates to prevent access or use by either the `Borrower` or the `Lender` during disputes. After the fee has been paid, the `Borrower` receives the access code to the bike, and the contract enters the `@Using` state.

2.4.5 Obligations

Stipula embodies another notable aspect of legal contracts: **obligations** that prescribe certain actions to be performed within a specified *timeframe*. These obligations are converted into commitments in *Stipula* and are evaluated when the time limit is reached, with the *event primitive* serving as the corresponding programming abstraction. For example, the foregoing `pay` function may be refined by issuing an event that terminates the renting service when the time limit is reached. The code becomes:

```
15    @Payment Borrower : accept[y]
16      (y == cost) {
17        y -o wallet
18        use_code -> Borrower
19        now + rentingTime >>
20          @Using {
21            "End_Reached" -> Borrower
22            wallet -o Lender
23          } ==> @End
24    } ==> @Using
```

The deadline for returning the bike is denoted by `now + rentingTime`, and if the bike has not been returned by then (i.e., the contract remains in the `@Using` state), a message requesting the bike's return is sent to the `Borrower` (line 7), and the fee stored in the wallet is transferred to the `Lender` (line 8). It should be noted that *events are not triggered by any party*; they are **automatically executed** when the time condition is met. Since the statements within an event's body will be executed in the future, it is assumed for simplicity that the event's body is outside the scope of function parameters, both for assets and non-assets.

2.4.6 Third party enforcements

Stipula offers a straightforward approach for modelling disputes without the need for additional features, which resembles the actions of a court. When the software is unable to verify contract violations, such as bike damage or misuse, or the rental of a

defective bike, a trusted third party, the `Authority`, must be involved in supervising the dispute and providing a resolution mechanism. The following code demonstrates how off-chain monitoring and enforcement mechanisms are encoded in *Stipula* through an `Authority`, which must be included in the agreement. The following code is just an example to show the functionality *third party enforcements* and external to the `BikeRental` code:

```
1   @Using Lender,Borrower : dispute(x) {
2     x -> _
3   } ==> @Dispute
4
5   @Dispute Authority : verdict(x,y)
6     (y >= 0 && y <= 1) {
7       x -> Lender, Borrower
8       y * wallet -o wallet, Lender
9       wallet -o Borrower
10  } ==> @End
```

The `dispute` function can be triggered by either the `Lender` or the `Borrower` and includes a string `x` as the reason for initiating the dispute. After the reasons are communicated to all parties (represented by `_` instead of writing out three sending operations), the contract transitions to the `@Dispute` state, where the `Authority` will analyze the issue and issue a verdict. This is accomplished by allowing only the verdict function to be invoked in the `@Dispute` state. The verdict function takes two arguments: a string `x` indicating the reasons for the decision and a coefficient `y` that represents the portion of the wallet that will be reimbursed to the `Lender`; the remaining portion will be given to the `Borrower`. It is important to note that the statement `y * wallet -o wallet` represents `Lender` receiving the `y` portion of the wallet (`y` being in the range [0...1]) and the wallet being adjusted accordingly. The remaining portion is sent to the `Borrower` with the statement `wallet -o Borrower`, which is shorthand for `1 * wallet -o wallet, Borrower`, and the wallet is then emptied.

Chapter 3

Analysis and design of Stipula platform

This chapter provides a possible design for the implementation of the *Stipula* language. In particular, the general architecture of the implementation will be illustrated, with its components and modules. Furthermore, the various design choices will be explained and justified.

3.1 Blockchain layers

Over the years, blockchains such as those of Bitcoin and Ethereum have established themselves in the new distributed context. One problem that has arisen and is recognized is the *scalability issue* of these solutions. Blockchains like the ones mentioned fail to scale much when faced with having to process a large amount of transactions. Furthermore, this amount of transactions can congest the network of blockchain nodes and increase fees for logging information into the ledger. Therefore, over the years, solutions have been designed and implemented that could remedy the problem of scalability. These new solutions use an *underlying blockchain as a basis*, such as Bitcoin and Ethereum, and develop solutions that allow for an increase in transaction throughput, while trying to keep the cost of commissions low. To do this, scalability solutions need a secure and decentralized layer. Networks such as Bitcoin and Ethereum are defined as **layer one**, that is, they represent the fundamental layer on which it is then possible to build other applications, while scalability solutions are defined as **layer two**.

There are mainly four types of layer two:

1. *Nested blockchain* (i.e., Plasma, *Plasma chains*): they run on top of another (i.e., Ethereum). Layer one establishes the settings and layer two conducts the procedures;

2. *Sidechains* (i.e., Polygon, *Polygon Technology*): delegate massive transactions, but this kind of chain is less integrated into the core blockchain. They can have a different consensus algorithm from the blockchain layer one;

3. *Rollups* (i.e., Arbitrum, *Arbitrum*, and Optimism, *Optimism*): the transactions are computed off-chain but the data are saved on the blockchain layer one;

4. *State channels* (i.e., LightningNetwork, *Lightning Network*): establish two-way communication between a blockchain layer one and off-chain channels. The transactions are saved on the blockchain layer one when they will be completed on the state channel.

The first three solutions use a dedicated ledger in layer two similar to that of layer one. Instead, state channels use other cryptographic methods and ad-hoc communication protocols, without the need to have their own ledger in which to store the information. Only a minimal amount of information, produced by this type of scalability solution, is stored in the underlying layer one.

This separation between layers allows:

1. Layer one to focus on network and ledger distribution and security;

2. Layer two to focus on *scaling* transaction processing.

3.2 Basic ideas

As stated in the dedicated chapter, it was possible to notice how the *Stipula* language offers peculiar functions compared to the other languages present in the smart contract environment. In particular, it has been possible to find that the language requires that the concept of time is not dependent on external factors, such as the time required for the generation of a block. Furthermore, the functionality of invoking events scheduled over time is such that it must be managed ad-hoc in the implementation, and that currently there is no similar one in the various languages for smart contracts, except by resorting to the use of software external to the system (i.e., oracles). Therefore, due to these particular functionalities that must be guaranteed by the language, one of the ideas behind the realization of the project is to provide an implementation of the language as a dedicated *layer* which rests on another underlying layer. With this concept we want to place a marked separation between the layer that deals with saving information in an immutable register and guaranteeing security, and the layer that deals with the execution of contracts. Thus subdivided, the lower layer is used as a *commitment* state, where the upper layers use that layer to *timestamp* the information; whereas, the layer that implements the *Stipula* language only deals with the execution of contracts (i.e., contract status management, asset transfer management, ...) and guarantees compliance with all the constraints imposed by the language , delegating information storage to the lower layer.

Another of the main ideas that guided the entire planning phase, and subsequently also the implementation phase, is the creation of a system that could be performed both in *centralized* and in *distributed* form. In fact, the implementation developed in this thesis work allows you to:

1. Use it as a single instance on a server;

2. Create a distributed network of *nodes*. By node we mean an instance of the implementation that is able to communicate with other instances of the same implementation to update itself regarding, for example, the states of the contracts in execution. A network of nodes thus defined can be centralized, that is, there is a single point of control that manages all aspects of the network (i.e., updates, communications between nodes, ...), or, *decentralized*, that is, not there is a central body that manages the network, but all the nodes are equal to each other

(a *peer-to-peer* network). By doing so, a network of nodes providing a language implementation can be classified as a layer one;

3. Create a network of nodes that rely on an underlying layer (i.e., HyperLedger Fabric). It is also possible to use layer two as a commitment layer. State channels (i.e. *Lightning Network*) could also be used for data commitment. However, the way they're designed doesn't make writing information as easy as other scaling solutions (see section 3.1). Again, network management can be centralized or decentralized.

Throughout the thesis, reference will be made to **Stipula server** and **Stipula node**: *server* means the instance of the language implementation running on a single machine. This mode mirrors the *client-server* architecture, in which there is a machine that provides a service and some clients who want to take advantage of this offered service. By *node*, on the other hand, we mean that the instance of the language implementation runs on different machines, which are able to communicate with each other to exchange information. Therefore, this architecture needs communication protocols and a layer that allows to manage the *consent*. In particular:

1. A *Stipula server* is a application that runs in a single machine. It is able to execute contracts and transfer assets among users. It does not have any module about consensus or module that allows a communication with other *Stipula servers*;

2. A *Stipula node* is able to do all the things cited at the previous point, and it is able to communicate with other *peers* (other nodes), in order to exchange information about the network and the consensus about contract states evolution.

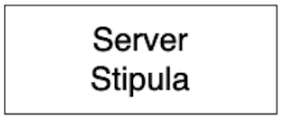
In the figure 3.2 it is possible to notice the possible configurations that the designed architecture can support:

1. It is possible to run *Stipula server* on a single machine. This server has no way to communicate with other servers and is suitable for a centralized concept;

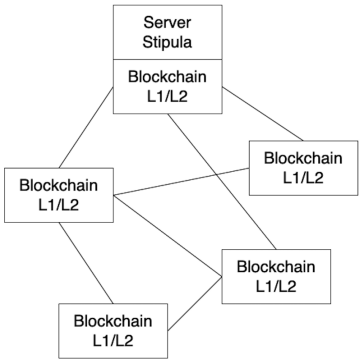
2. It is possible to run a *Stipula server* and which uses a blockchain as an underlying layer. The blockchain can be both layer one (**L1**) and layer two (**L2**) as long as it is possible to record information in a ledger (therefore it is not possible to use state channels). All information regarding contracts and asset transfers will also be stored in the underlying layer;

3. It is possible to create a network of *Stipula nodes*, which are able to communicate with each other for the maintenance of the network itself and to exchange information regarding the consensus of the results obtained from the execution of the contracts. There is no need for an underlying layer;

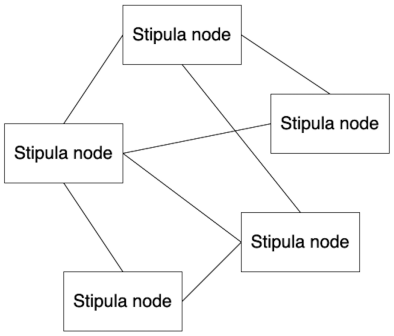
4. To increase security, it is possible to extend the scenario described in the previous point and also use a blockchain as a lower layer in which information regarding contracts and asset transfers can be immutably stored.



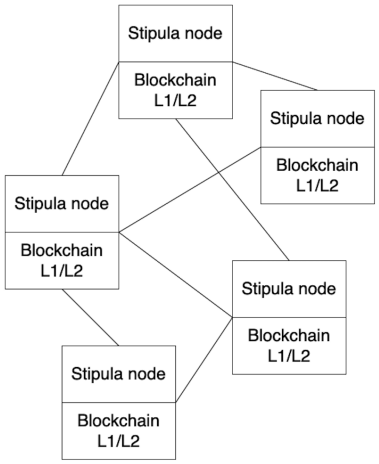
(a) Example of an instance of a *Stipula* server.



(b) Example of an instance of a *Stipula* server that relies on a blockchain.



(c) Example of a network of *Stipula* nodes.



(d) Example of a network of *Stipula* nodes based on a blockchain.

Figure 3.1: Possible configurations of the *Stipula* architecture.

3.3 Architecture

The main modules that make up the implementation architecture of the *Stipula* language are (3.2):

1. *Message Service*: manages connection and communication with clients;
2. *Compiler*: receives as input a contract written in the *Stipula* language and compiles it in another language, called *Stipula bytecode*;
3. *Virtual Machine*: execute the code in *Stipula bytecode* received as input;
4. *Consensus*: implements mechanisms that make it possible to determine consensus regarding the result of executing a contract, within a network of nodes;
5. *Storage*: stores information about *assets* and its *transfers*, *contracts* and its *instances*;
6. *Commitment*: deals with communicating with the layer that allows you to securely store some of the information stored in the *storage* layer. At this level, it is not necessary to save exactly all the information stored in the storage, it is sufficient to save a minimum set of such information;
7. *Communication protocols*: implements a series of protocols necessary for the consensus layer and for communication between nodes (i.e. *node discovery*).

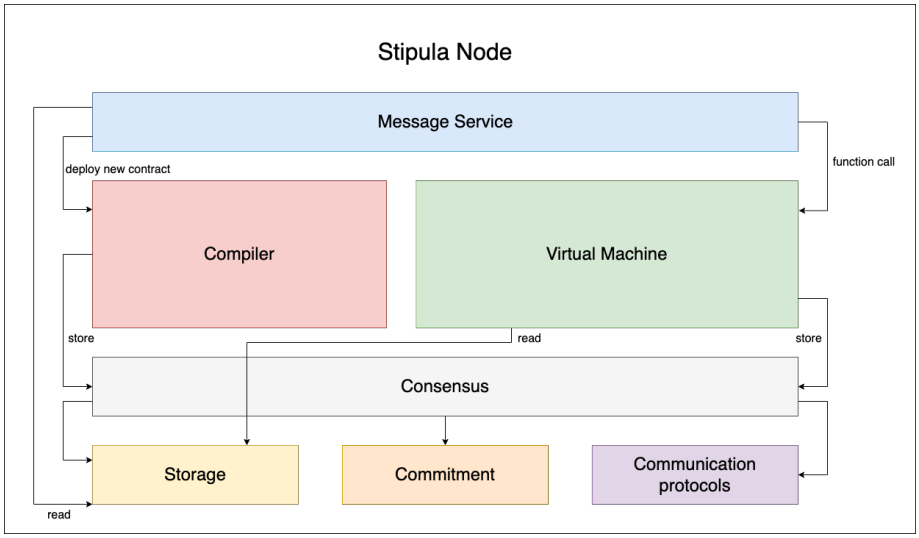


Figure 3.2: Complete architecture of all modules and interactions between them.

3.3.1 Message Service

This module performs the role of *server*, that is, it waits for new connections from clients. It was decided to use a TCP connection instead of HTTP for the following reasons:

1. In order to communicate via the HTTP protocol, a *web server* must be added to the implementation. This would have weighed down the implementation and would not have made use of all the features offered by the HTTP protocol;

2. The TCP protocol is a *reliable* and *connection-oriented* protocol, that is, it frees the application from the task of handling out-of-order or missing packets;

3. The TCP protocol allows you to send and receive streams of bytes directly, instead of streams of characters as is the case with the HTTP protocol. By doing so, you also get an advantage in terms of efficiency.

Once a new connection has been received from a client, the management of communication with that client is delegated to a *thread*, in order to free up the server thread to accept new requests. Once the message sent by the client has been received and decoded, we proceed to check the *signatures* of the message itself. Every message that is sent must be signed with the private key held by the client. If the signature check passes, then the request will be routed to other modules. The connection with the client remains open until the module to which the request was directed returns a response; once the response is received, it is sent to the client and the connection is closed.

3.3.2 Compiler

A first implementation of the *Stipula* language has been done previously. In this implementation, the language is *interpreted* at *runtime* and is executed locally on the machine. This approach has limitations if you want to translate it into the distributed context. Interpretation has the advantage of directly executing the code, without performing intermediate compilation steps. However, it has major drawbacks:

1. *Slow to execute*: the code is translated directly into machine code. An intermediate compilation step could have optimized the code before running it;

2. *Low performance*: since the code is not optimized for the platform on which it runs;

3. *Less error checking*: this represents one of the weakest points of this approach, particularly in the context of writing and executing contracts that have legal value.

To overcome the limitations of this first implementation, it was decided to divide the process of executing a contract into two main steps:

1. *Compilation*: having received the contract as input, it is compiled in an intermediate language called *Stipula bytecode*. The characteristics of this language will be extensively discussed in the following chapter;

2. *Execution*: having received the contract compiled in bytecode as input, this is executed.

In this way we obtain an optimization in terms of performance, that is, once the contract has been compiled, the same contract can be called numerous times, without having to re-compile each time. Furthermore, having an intermediate compilation step it is possible to perform *static* checks: it is possible to verify whether the contract states can actually be reached during execution, check if the asset transfers have been defined correctly and implement checks about the types of variables. As regards this last point, the *Stipula* language is a *weakly typed* language: when writing a contract it is possible to specify whether a variable is a `asset` or is a `field`. Obviously, if a variable is defined as `field` it is necessary to perform a *inference* operation to determine the specific type. However, it's not always possible to determine the type of a variable at compile time, so you need to determine it at runtime.

The use of the bytecode language also allows you to obtain an advantage from the point of view of *interoperability*. In fact, further compilers could be developed, which from other smart contract languages (i.e., Solidity), translate the smart contracts into *Stipula bytecode* language.

3.3.3 Virtual Machine

The compilation phase produces code written in a particular language, specifically defined for the purpose of the project. Therefore, you need to build an environment for executing the contract bytecode. For this reason a *virtual machine* has been developed. The benefits that follow from virtualization are:

1. *Platform independence*: the virtual machine creates an abstraction layer between the code and the underlying hardware, thus allowing the code to be executed on different platforms, without the need to recompile it every time;

2. *Secure environment*: the virtual machine creates a secure environment for execution, especially needed for asset transfer.

The virtual machine implemented is a *stack-based virtual machine*, that is, the program variables are loaded onto a stack and the language instructions operate on that stack by removing or adding elements to the stack. A stack-based virtual machine is simpler to implement and requires fewer instructions to perform the same operations compared to a *register-based* virtual machine. However, it must be taken into consideration that a stack-based virtual machine is less performing than a register-based virtual machine and also requires many more memory accesses. Aware of these differences, due to issues of time and complexity, it was decided to create a stack-based virtual machine.

The virtual machine manages various memory areas, as it must take into account the internal variables of the functions, the parameters of the functions and the global variables. In addition to these, there are also other memory areas whose purposes will be defined in the next chapter.

This module also takes care of managing the *obligations*, that is, scheduling the events that will allow you to execute a specific portion of code, at a specific time. The management of this functionality involves both the compiler, in recognizing the obligation encoded in the code, and the virtual machine: during the execution of the contract, the scheduling request of an obligation is managed ad-hoc by a specific component of the virtual machine.

3.3.4 Consensus

Consensus is an essential component for the proper functioning of peer-to-peer networks, as it allows multiple nodes to agree on a single *result*, even in the presence of failures. Without consensus, peer-to-peer networks would be prone to conflicts, inconsistencies, and vulnerabilities, making them less reliable and less secure. In a distributed context, multiple nodes work together to perform a task and must communicate with each other to exchange information and coordinate their actions. However, due to factors such as network delays, failures and communication errors, different nodes may have different views of system status (such as a contract or a transfer) or may produce conflicting results. In particular, the goals of consensus algorithms must guarantee:

1. **Data consistency**: in a distributed ledger, multiple nodes can contain different copies of the same data. Consensus algorithms ensure that nodes agree on a single version of the data, ensuring data consistency and avoiding data corruption;

2. **Fault tolerance**: nodes can fail or become unresponsive. The consent algorithms allow the system to continue to operate even in the presence of node failures;

3. **Conflict-free**: consensus algorithms ensure that the system is conflict-free, ensuring that nodes agree on a single result;

4. **Security**: Consensus algorithms can help prevent malicious actors from manipulating the system by ensuring that all nodes agree on a single decision value, even in the presence of attacks such as data tampering or *Denial-of-Service* (*DoS*).

Therefore, the evolution of the status of a contract or the successful transfer of a certain amount of assets towards an address must be in agreement with the other nodes making up the network.

3.3.5 Storage

This module takes care of storing various information regarding assets and contracts. In particular, information concerning:

1. **Asset**: it is necessary to memorize information that characterizes the asset itself, such as the name, a unique identifier, the total *supply* and whether the asset is divisible or not. Total supply is the total amount of asset that will be available for a specific asset;

2. **Transfer of assets**: it is necessary to trace the movements of assets that are carried out by customers and contracts. Later it will be explained in detail how asset transfers occur in the current implementation;

3. **Contracts**: the new contracts are loaded into a *Stipula server* or a *Stipula node* and stored together with the bytecode produced by the compilation phase. Once a contract is uploaded, it is no longer possible to make any changes;

4. **Contract instances**: when you want to execute a contract, a new *contract instance* is created. The storage keeps track of *each running contract instance*, together with its *current state*. For instance, two actors, Alice and Bob, want to activate an instance of BikeRental with ItalyRent (see section 2.4.1). There can be a BikeRental between Alice and ItalyRent in state `@Using`, and a BikeRental between Bob and ItalyRent in state `@Return`.

3.3.6 Commitment

For the context created by the proposed implementation, the commitment module represents the level at which information can be securely stored. Indeed, the purpose of this component is to offer a high level of security for the recording of information. It is not necessary to memorize all the information that is saved in the *Storage* module, but a *minimal* set of key information allows to reconstruct the evolution of the contract and of the asset transfers that have taken place over time. Thus, the commitment module is used to *timestamping* the information, proving the *existence* of a particular piece of information. The presence of this module within the implementation is not strictly necessary for the functioning of a *Stipula* server or node, but it allows it to offer a higher level of security. As an example: we can devise a simple client-server implementation where the storage in managed by the server as an internal database, or a richer implementation where the server commits to a blockchain part of the storage to notarize the main info of the contract execution.

3.3.7 Communication protocols

In a distributed context, it is necessary that the various nodes can communicate with each other to exchange information. The type of information exchanged can be divided into:

1. Information to determine *consent* about the status of a contract;
2. Information to manage *connections* with other nodes (i.e., discovery of new nodes, notify that a node is congested, ...).

The design of a communication protocol is critical because it determines how efficiently and accurately information is transmitted and received. Poorly designed protocols can cause a variety of problems, such as slow performance, lost or corrupted data, security vulnerabilities, and difficulties in scalability and maintainability.

3.4 Interaction between modules

In this section we want to explain how the interactions between the various modules of the architecture take place. Each request sent by a client is received by the *Message Service* module, which, after carrying out the appropriate checks, can direct the request towards different modules according to the specific request:

1. Towards the *Compiler*: when the request received represents the user's will to load a new contract. The contract is received from this module, which proceeds with the compilation of the same;
2. Towards the *Virtual Machine*: when with this request the client wants to create a new instance or perform a function of a specific contract instance.
3. Towards the *Storage*: when the client's request consists in a reading of some information (i.e., available assets associated with its address).

It is necessary to specify that all the requests that are addressed to the *Storage* module are always read requests and never write requests; writing information can only be done for compiling a new contract (*Compiler*) or for running an instance of a contract (*Virtual Machine*).

If the request is of type (1) or of type (2), the results produced by the respective modules are routed to the consent module. The reason is that before writing any information to the storage and/or commitment layer, the network must agree on the same information to be written. Consequently, this phase also involves the module that implements all the communication protocols between the nodes, since, in order to be able to determine consensus within the network, the nodes must be able to communicate with each other. Furthermore, this module will carry out activities regardless of the requests received from the clients, in fact within this part of software there are also the protocols that allow you to manage the connections with the other nodes.

3.5 Asset management

The language offers primitives that allow you to handle the transfer of assets carefully and independently of how the assets are actually implemented. These design choices made the language very clear to write and read code.

3.5.1 Definition of assets

One of the most important points of the development of this thesis project concerns the provision of an implementation of the concept of *asset*. As a basic idea, it was decided to try to reproduce the same characteristics of blockchain *token* such as *Ethereum* or *Algorand*, adapting the complexity to the current development of the project. Thus, the main characteristics of an asset are:

1. **Unique identifier**: consists of an alphanumeric string to uniquely refer to an asset;
2. **Name of the asset**: a name is defined that can be easily remembered by a person;
3. **Unit name**: corresponds to what is a *ticker* of a company listed on the stock exchange (i.e., APPL for Apple company);
4. **Decimals**: indicates how many parts a single unit can consist of;
5. **Maximum supply**: indicates the maximum amount of assets that can exist over time. The maximum quantity also includes decimal values, for example: if you want to define an asset `StipulaCoin` which can only have 10 units and each unit can be divided into 100 sub-units, therefore the maximum supply will be 1000 sub-units.

Example of definition of an hypothetical `StipulaCoin` asset:

1. *Unique identifier*: `stipula_coin_asd345`
2. *Name of the asset*: `StipulaCoin`
3. *Unit name*: `STC`
4. *Decimals*: 3
5. *Maximum supply*: 1000

The `StipulaCoin` asset thus defined has an identifier (`stipula_coin_asd345`), a unit name (`STC`), has a maximum supply of 1000 sub-units and the number of decimals is equal to 3, so a single unit can be divided into 100 sub-units.

With this structure it is possible to define different types of assets, such as:

1. **Fungible assets** (i.e., bitcoins and banknotes): an asset is fungible when it is *interchangeable* from the point of view of the units that compose it, i.e., that each of its units is *indistinguishable* the from each other, for the same nominal value. It is possible to define both divisible and non-divisible fungible assets;

2. **Non-fungible assets** (i.e., NFT): contrary to fungibility, the units that make up the asset are unique and therefore are not interchangeable with each other. Furthermore, another feature that distinguishes them from fungible assets is that they are not divisible.

An example of a fungible asset was shown above (3.5.1). An example of a non-fungible asset is provided:

1. *Unique identifier*: `stipula_nft_abc123`

2. *Name of the asset*: `StipulaNFT`

3. *Unit name*: `SNFT`

4. *Decimals*: 0

5. *Maximum supply*: 1

The `StipulaNFT` asset thus defined has an identifier (`stipula_nft_abc123`), a unit name (`SNFT`), has a maximum supply of 1 unit and the number of decimals is equal to 0, because a non-fungible asset cannot be split.

However, the definition of the characteristics of an asset is not sufficient to also define in which *way* the assets are transferred. Therefore in the next section we will proceed to provide a possible solution for asset transfer.

3.5.2 Transfer of assets

The way in which the transfer of assets is implemented represents one of the most delicate points of the whole project. When you want to send a certain amount of a specific asset, you want to guarantee some properties:

1. **Atomicity**: the transaction must take place atomically from the sender to the recipient;

2. **Consistency in the quantity transferred**: we want to ensure that no quantities of assets are lost or quantities of assets are generated out of nothing, during a transaction;

3. **Prevent assets from getting stuck**: you want to prevent assets from getting stuck in contracts, and therefore cannot be spent;

4. **Proof of possession**: you want to have proof that the amount of assets you want to move is actually owned by the sender;

5. **Avoid double-spending**: prevent an entity from being able to pay two recipients using exactly the same amount of assets. This is a problem that does not arise with, for example, banknotes, but it is a problem that affects digital assets. This problem is due to the fact that in the digital context, unlike the real world, it is difficult to reproduce the concept of *scarcity* of a good.

The problems indicated in points (1) and (3) are totally solved by the *Virtual Machine* (see section 4.6), however, the check that during the transfer of assets no amount of assets is lost or generated, is only solved partially (see sections 4.1.2 and 4.6.3). The solution to the other points will be explained later in sections 4.1.2 and 4.6.

Pay-to-Contract and Pay-to-Party

Blockchains like *Ethereum* or *Algorand* allow you to send and receive coins without using contracts and to exchange tokens using smart contracts. In the current implementation for *Stipula*, it is not possible for two entities to exchange assets except through a contract. This is because it would have required the development of primitives external to the virtual machine in order to enable the transfer of assets without going through a contract. The purpose of the implementation is to build a platform for the execution of legal contracts, and not to create a platform for pure asset transaction.

Taking into account that any transfer of assets must take place through the execution of a contract, it is necessary to define how these transfers must take place between the participants of a contract. Each *party* of a contract (that is, the participants of the contract) owns a pair of *cryptographic keys*, with which it is able to send signed messages and to prove possession of certain properties. A party, not being able to directly send assets to another party, must send these assets to the contract. This operation is defined as **Pay-to-Contract** or **deposit**, i.e., when the party makes a function call that requires sending a certain amount of assets, it sends it to the contract. In the execution phase of the function call, all the necessary checks will be performed to verify that the party is actually the owner of that amount of assets. Once the checks have been carried out, it will now be the contract that guarantees the integrity of the assets, i.e., thanks to the definition of the language primitives, it will be the contract that ensures that no quantities of assets will be lost into thin air or quantities of assets will be generated from nothing . Instead, the operation that occurs when the contract has to send assets to a party is called **Pay-to-Party** or **withdrawal**. Again, the language primitives will ensure that the party will receive the correct amount of assets. Thus defining the deposit of assets in a contract and the withdrawal of assets from a contract, it is possible to transfer assets between two or more parties by means of a contract, without the need to implement additional primitives external to the language and specific to the implementation of the architecture. A simple example that illustrates how *Pay-to-Contract* and *Pay-to-Party* work is shown in the example in section 4.8.1, where two actors, Alice and Bob, exchange two assets with each other.

Single-use-seals

Despite the definition of an asset management structure and the definition of the operations to send and receive assets, the definition of a model to represent and manage the balances of various assets that a party can have is missing. The simplest model to implement is the **account-balance-based** model, where each address has a balance associated with it. With each transaction, the balance of the sent asset is

updated, both for the sender and for the recipient. An example of an account-balance-based model:

```
...
"partyA": {
        "asset1": {
                "balance": 123.65
        },
        "asset2": {
                "balance": 18.44
        }
}
...
```

However, this simple model has problems in terms of:

1. *Scalability*: if you want to send two transactions of an asset `asset1`, these transactions cannot be parallelized, as the second transaction must wait for the balance of asset `asset1` is updated for the sender and the recipient after the execution of the first transaction;

2. *Privacy*: it is a model that allows you to very easily track the funds associated with an address.

An alternative model has been introduced by Bitcoin: the **Unspent Transaction Output** (**UTXO**) model. It is a more complex model than the previous one but offers advantages in terms of scalability, privacy and security. To give a concrete example, we can define a UTXO as a box that contains a certain amount of an asset. This box is closed by a *single-use-seal* which can only be broken by the owner of the quantity of assets in question. To explain how this model works, we introduce the following example: suppose Alice has to give Bob 1 `StipulaCoin`. Alice owns two UTXOs, `UTXO_1` and `UTXO_2`, each of 1 `StipulaCoin`. At this point, Alice chooses to use `UTXO_1`, she breaks the seal and creates a new seal so that only Bob can then break it in turn. By doing so, `UTXO_1` it becomes Bob's property and only he can spend it in future transactions. The breaking of the seal can only be done if the user is able to provide cryptographic proof that he is the rightful owner of the UTXO. The balance of Alice and Bob's **wallets** are given by the sum of the UTXOs in their possession: before the transaction, Alice had 2 `StipulaCoin` contained in two different UTXOs, that is, *two transaction outputs (received) not (yet) spent*, while Bob had 0 `StipulaCoin`; after the transaction, Alice has 1 `StipulaCoin` and Bob has 1 `StipulaCoin` (the one received from Alice), that is, both have *an output of a transaction (received) not (yet) spent*.

The advantages of using this model are:

1. *Parallelization*: unlike the account-balance-based model, transactions that correspond to two payments using two different UTXOs can be parallelized, without having to wait for the balance to be updated of the transacted asset after the first transaction. Taking the above example, Alice could send `UTXO_1` to Bob with a transaction and send `UTXO_2` at the same time Charlie;

2. Partially solves the *double-spending* problem: a specific UTXO can be spent on only one transaction and not on others. The problem is partially solved because the network of nodes still has to verify that a user does not try to pay multiple transactions with the same UTXO;

3. *Security*: in order to spend funds, cryptographic proof of ownership of the assets to be moved must be provided;

4. *Privacy*: this model encourages not to reuse the same addresses, making it more difficult to trace funds. In fact, each transaction accepts UTXO as input and generates new UTXO as output. If the same address is used over and over again to make payments, it becomes easier for an observer to track the history of transactions associated with that address, potentially revealing sensitive information;

5. **UTXO selection**: consists of the process of choosing the UTXOs, from the set of available UTXOs, to cover the transaction amount while minimizing the transaction fees in Bitcoin. If you select UTXOs with a large value, you may pay higher transaction fees than if you selected UTXOs with a smaller value. Also, if you select too many UTXOs to cover your transaction amount, you may find yourself having a larger transaction size, which can also result in higher fees. Introduce the following example to clarify the implications of this technique: suppose you want to send someone 0.5 `StipulaCoin` and that you have several UTXOs in your wallet, including:

   * 1 `StipulaCoin`
   * 0.7 `StipulaCoin`
   * 0.5 `StipulaCoin`
   * 0.2 `StipulaCoin`

   If you select the UTXO from 1 `StipulaCoin` to complete the transaction, you will end up paying a higher commission than if you selected the 0.5 UTXO `StipulaCoin`. This is because the transaction size for the UTXO is 1 `StipulaCoin` is greater than that of the 0.5 `StipulaCoin` UTXO, which means that it will cost more to include in a block. Therefore, selecting the appropriate UTXOs for a transaction is important to minimize fees and ensure that the transaction is processed quickly and efficiently.

For the implementation of the *Stipula* language, it was decided to pay greater attention to network security (avoiding double-spending), user security (verifying ownership of funds through cryptography) and privacy. Hence, it was decided to implement a model similar to that of Bitcoin. The implemented model representation is much simpler than that of a UTXO, therefore to keep separate the original concept from the one implemented for the thesis, it was decided to use a different name: **single-use-seal**. This term wants to refer in particular to the key action that occurs during a transaction, i.e. the breaking of the seal by the sender and the creation of a new seal that only the recipient will be able to break in turn. This step represents the *transfer of ownership* of a certain amount of assets from one user to another.

Script

The *smart contract* concept was first proposed by cryptographer Nick Szabo (Nick Szabo, 1997). Szabo described smart contracts as computer protocols that facilitate, verify, or enforce the negotiation or performance of contractual obligations, without the need for a trusted third party. However, it was only with the development of blockchains that smart contracts became practical to implement. The first blockchain-based platform where it was possible to create and execute smart contracts was

Ethereum. This blockchain introduced a programming language called *Solidity*, which allows developers to write smart contracts and deploy them on the blockchain.

Before Ethereum, the first large-scale application of the smart contract concept was Bitcoin with the implementation of the **Script** language. It is a *stack-based* language and offers a flexible and secure way to define the conditions under which a transaction can be spent in the Bitcoin network, enabling more advanced transaction types, such as requiring more signatures or a certain amount of time before that the funds can be transferred. These rules can be combined in different ways to create more complex transaction types and smart contracts. For example, a transaction with multiple signatures might require approval from multiple parties before funds can be transferred, while a time-locked transaction might require a certain amount of time before funds can be accessed.

In the previous section we introduced *UTXO* and its simplified version, *single-use-seals*. When a user has to pay for a contract, he has to provide cryptographic proof of ownership of the single-use-seal. The naive solution is to send a message representing the call of a function, within which the cryptographic demonstration is provided. The cryptographic proof can consist in signing the single-use-seal identifier with the private key, so that the user can prove possession of it. However, this solution is very limited: if you wanted to implement a payment that requires approval from different users, you would have to update the message format or create a new message type (i.e., update the `FunctionCall` message in order to collect more signatures). To avoid having to create many different message formats, it is possible to encode the cryptographic proof as a contract written in *Script*. By doing so, the cryptographic proof can be represented by a complex contract written in *Script* and which will be validated to verify if:

1. The program is syntactically and semantically correct, and

2. The signatures collected are correct and the conditions imposed by the contract have been respected.

Szabo's smart contract idea is different from *Stipula*'s legal contracts. *Script* allows you to manage the expenditure of funds in a secure and advanced way, *Stipula* takes care of executing programs that codify specific contractual obligations. The two languages have two different purposes and therefore can coexist in the same implementation. In fact, a language similar to *Script* was used to manage asset transfers and whose programs are executed by a specific component of the virtual machine. Thus, the *Virtual Machine* module allows both to execute contracts in *Stipula* and contracts in *Script*. By analogy with the language used in Bitcoin, it was decided not to change the name of this language, as they share the same instructions and mechanisms. All the details of how the *Script* language works will be described later.

Fungibility of assets and non-fungibility of single-use-seals

In the 3.5.1 section, the difference between fungibility and non-fungibility has been defined. It should be noted that a UTXO or single-use-seal *is not fungible*. This is a property that both UTXO and single-use-seals have in common, so during the explanation we will refer to UTXO for the sake of brevity. A UTXO is non-fungible as it represents the value (quantity of assets) of a specific output obtained from a specific transaction. Once a UTXO is spent in a transaction, this UTXO is *consumed*, and therefore can not *never* be used again for a future payment. Also, a UTXO cannot be split, that is, you cannot break the seal, take a fraction of the amount of assets it

contains, and put the same seal back on. If Alice has to send 2 `StipulaCoin` to Bob but she only has a UTXO of 5 `StipulaCoin`, the following actions will happen:

1. Alice provides cryptographic proof that she owns the UTXO containing 5 `StipulaCoin`, so she breaks the seal for that UTXO;

2. Alice prepares two new UTXOs: one of 2 `StipulaCoin` with a seal that only Bob can break, and a 3 `StipulaCoin` UTXO with a seal that only Alice can break. The latter UTXO represents the *remainder* of a transaction. A similar situation can be encountered when using banknotes: if you have a banknote whose value exceeds that of the item you want to buy, the seller will withhold the amount equal to the value of the item and return it to the buyer the difference.

The non-fungibility of UTXOs is important because it ensures that each transaction is recorded as a unique event, with a specific sender, recipient and value. This provides a high level of transparency and security, as it is difficult to manipulate transaction history. Also, since UTXOs cannot be reused, it reduces the risk of double-spending and makes it easier to track the movement of funds. However, it also means that it can be more difficult to make small or precise transactions, as UTXOs must be screened in their entirety.

Difference from Ethereum

When a user wants to interact with a smart contract on Ethereum, the user must authorize it to access their funds. By doing so, the contract will be able to spend funds in your name. This practice is called *token allowance*. This feature represents a serious weakness regarding the security of funds. Indeed, there have been several cases of users approving malicious contracts with the aim of exfiltrating all possible funds. The philosophy which has been pursued for the implementation of the *Stipula* language, and which has already been introduced previously, consists in giving the user full control over his own funds. When a user wants to make a function call of a specific *Stipula* contract that requires the sending of a certain amount of assets, the user cryptographically proves ownership of the single-use-seal to be sent and transfers ownership to the contract. Now that the funds belong to the contract, the latter will be able to manage them adequately according to the defined rules. A contract *Stipula* can never embezzle users' funds without their explicit permission. Obviously, this approach has the disadvantage for the user of having to sign several transactions if the contract requires it. Instead, in Ethereum, thanks to the token allowance, the smart contract can carry out several transactions, without having to ask the user to sign them each time. The proposed solutions are orthogonal to each other and it was decided to always aim to ensure a higher level of security, to the detriment of a limitation of the functions that can be offered.

Chapter 4

Implementation

In the previous chapter the general architecture of the project was introduced (figure 3.2). However, the current version is a simplified version of the initial architecture. In particular, the current architecture is illustrated in figure 4.1. The implemented architecture consists of a Java application, usable remotely via socket communication. All interactions with this architecture take place according to the classic *client-server* model. The reason for this simplification is mainly due to the complexity of developing components, such as the *Virtual Machine*, the compiler and the asset management model (*single-use-seals*). However, the design of this architecture also takes into account future developments, which will be illustrated in the next chapter. The current architecture represents a starting point for the realization of the architecture presented in the previous chapter.

4.1 Introduction to basic concepts

The previous chapter introduced the concepts that form the basis on which the current architecture is based, such as assets and their management, and contracts. In this section these concepts will be analyzed from an implementation point of view, in order to help in understanding the functioning of the architecture in its complexity.

4.1.1 Contracts and contract instances

In the current version of the architecture, there is a difference between a *contract* and a *instance of a contract*. The difference is similar to *class* and *object* (or *instance of a class*) for object-oriented programming. When two actors want to *execute* a contract, they will agree to create a new instance of a contract. The contract will be *immutable*, while a contract instance can change over time.

From an implementation point of view, a contract is represented as follows:

1. The source code of the contract;
2. The compiled contract, that is, the bytecode;
3. The initial state of the contract state machine;
4. The final states of the contract state machine (optional);
5. The contract state machine transitions.

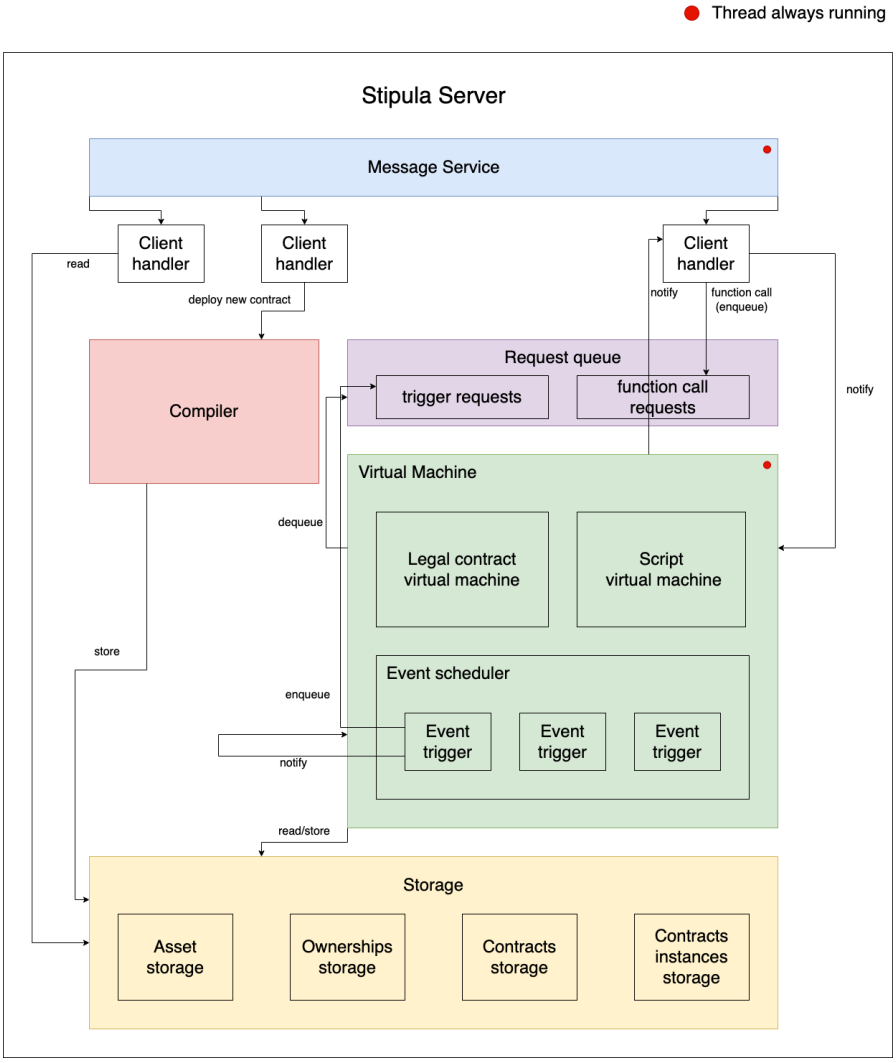


Figure 4.1: Current state of the implemented architecture.

When this set of information is sent to the *Storage* module, the latter generates an identification code to be associated with the contract. This identifier is sent in response to the client requesting to load the contract (see 4.3.4).

An instance of a contract is represented as follows:

1. The **identifier of the contract**: it must be specified which specific contract you want to refer to;

2. The **participants of the contract**;

3. The definition of a **memory space** dedicated to maintaining the state of the global variables during the evolution of the instance of the contract;

4. A **state machine**: this structure is needed to track the progress of the contract instance over time and to ensure that the contract participants operate without violating the established order of operations.

This distinction between contract and instance of a contract allows for the creation of multiple instances starting from the same contract, i.e., multiple users can use the same contract multiple times, present in a server or *Stipula* node, creating multiple instances.

4.1.2 Asset

Definition

As illustrated in the previous chapter (see 3.5.1), the goal is to try to reproduce the concept of *token*, as is the case for Ethereum. An asset within the architecture is represented in Java by the object `AssetConfig` class:

1. `String assetName`: a name is defined that can be easily remembered by a person;

2. `String unitName`: corresponds to what is a *ticker* of a company listed on the stock exchange;

3. `int decimals`: indicates how many parts a single unit can be in;

4. `int supply`: indicates the maximum amount of assets that can exist over time.

When this set of information is sent to the *Storage* module, the latter generates an identification code to be associated with the asset. The object being stored contains the `String id` fields and `AssetConfig asset`.

At this point, creating *fungible* and *non-fungible* assets consists in extending the `AssetConfig` class. In particular:

1.
```
1  public class FungibleAsset extends AssetConfig {
2      public FungibleAsset(String assetName, String
       ↪  unitName, int supply, int decimals) {
3          super(assetName, unitName, supply, decimals);
4      }
5  }
```

2.
```
public class NonFungibleAsset extends AssetConfig {
    public NonFungibleAsset(String assetName, String
    ↪ unitName) {
        super(assetName, unitName, 1, 0);
    }
}
```

4.2 Libraries

This package contains all the fundamental data structures for the overall development of the project. Furthermore, a library that implements cryptographic functions has been implemented.

4.2.1 Crypto

This library implements a number of cryptographic features, useful both for the architecture and for external software such as SDKs and wallets. The implemented methods are:

1. `generateKeyPair`: allows you to generate a 1024-bit RSA key pair;
2. `encrypt`: allows you to encrypt the received input;
3. `decrypt`: allows to decrypt the received input;
4. `getPublicKeyFromFile`: allows you to create a public key from a file;
5. `getPrivateKeyFromFile`: allows you to create a private key from a file;
6. `readKeyFromFile`: allows you to read a key from a file;
7. `getPublicKeyFromString`: allows you to create a public key from a string;
8. `sign`: allows you to sign the received input;
9. `verify`: allows you to verify if a signature is valid.

4.2.2 Data structures

This package contains all the fundamental data structures for the implementation of the architecture. In particular, the data structures are:

1. `Pair`: represents a collection of two items of any type. The order of the elements is important and allows two related values to be stored and manipulated as a single element;
2. `Triple`: it is a structure similar to the previous one, but it allows to manage three elements;
3. `Queue`: is a data structure that implements the *First-In-First-Out* (*FIFO*) policy, ie, the first item added to the queue is the first to be removed. This structure is used when algorithms need to process a sequence of elements in a specific order;
4. `Stack`: is a data structure that implements the *Last-In-First-Out* (*LIFO*) policy, i.e., the last element added to the stack is the first to be removed.

In addition to these data structures, data structures provided directly by Java have been used, such as *ArrayList* and *HashMap*.

4.3 Message Service

This module is responsible for managing communication with clients, accepting their requests and redirecting them to the appropriate architecture modules. Before accepting requests, checks are carried out on the correct format of the message and the signatures associated with the message itself.

4.3.1 MessageServer

This component allows you to create an instance of a *server*, which waits for new connections from *clients*. When a new request arrives, the connection is delegated to a dedicated thread; by doing so, the server is ready to accept new connections. The other tasks of this component are to:

1. Instruct the dedicated thread, passing it all the objects it needs;
2. Allocate a specific zone in **shared memory**. This memory zone is shared between these threads and the virtual machine and is required for communication between these two components. From the point of view of the implementation, shared memory is represented by one *map*, where the key is a string that serves as an identifier to access the cell, and the value is a generic `T` object.

4.3.2 ClientHandler

This component takes care of managing a single connection with a client. In addition, this component takes care of:

1. Check the *signatures* of the message received from the client;
2. If the previous check is successful, this component takes care of directing the request to the correct module.

When a response has been received from the module to which the request was directed, the `ClientHandler` *deallocate* the memory zone from shared memory.

4.3.3 ClientConnection

This component allows you to manage the connection more easily. In fact, it exposes high-level functionality, hiding certain complexities regarding socket management. This component allows you to make the `ClientHandler` code more compact and readable.

4.3.4 Messages

The messages currently in the implementation will be explained below. These represent the fundamental messages to allow the execution of the contracts. In the future, this ensemble will certainly be expanded. For ease of implementation, message transmission consists of direct encoding of Java objects in JSON format.

DeployContract

This message allows you to load a new contract into the *Stipula* instance. The only required value is the *source code* of the contract. This request will then be routed to the compiler.

FunctionCall

This message allows you to make a function call for a specific instance of a contract. The required parameters are:

1. `contractInstanceId`: identifier of the instance of the contract to which it refers;
2. `functionName`: name of the function to call;
3. `arguments`: the list of arguments of the function to call. The elements of this list are *triples*. In this case, the meaning of a triple is *variable type*, *variable name* and *variable value*.

This request will then be routed to the virtual machine.

Pay-to-Contract In the previous chapter the concept of *Pay-to-Contract* was introduced (see 3.5.2), that is, the user can make a payment to an instance of a contract, using one of its *single-use-seals*. Previously, the `FunctionCall` object was introduced, which allows you to supply the parameters of a specific function. These parameters are specified by the `arguments` field, which is of type `Triple<String, String, Object>`. The last component of the triple accepts a value of type `String` or of type `PayToContract`. This last object allows you to provide all the information necessary to make the payment to the contract instance. In particular, the fields of the object are:

1. `String ownershipId`: it is the identifier of the *single-use-seal* that the user wants to spend;
2. `String address`: the address of the owner of the *single-use-seal*;
3. `String unlockScript`: consists of a cryptographic proof proving that the user is the effective owner of the *single-use-seal*. The meaning of this field will be described later.

Here is an example of *Pay-to-Contract*:

```
1    ...
2    "arguments": [
3      {
4        "argument": {
5          "first": "asset",
6          "second": "y",
7          "third": {
8            "ownershipId": "2b4a4614-3bb4-4554-93fe-c034c3ba5a9c",
9            "address":
               ↪  "ubL35Am7TimL5R4oMwm2OxgAYA3XT3BeeDE56oxqdLc=",
```

```
10            "unlockScript": "PUSH str PLjodnT+m3RNIitQAPBDCsRmJPHC
               ↪  qrwZOY/CPiHFZGnl+DRN6soqxMy3ehTFaUwxBjjf7qfBfvTDq5
               ↪  oBItTFrtz1Rn5SDS1ybdbkwpKaOXVglNOw7ZEG9bbZ1mo1oA7I
               ↪  AjRiIilzUetCstE5rPZIf9XOXr/RQ5AHkZUn2CztsvA=\nPUSH
               ↪  str MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCo/GjVKS
               ↪  +3gAA55+kko41yINdOcCLQMSBQyuTTkKHE1mhu/TgOpivM0wLP
               ↪  sSga8hQMr3+v3aR0IF/vfCRf6SdiXmWx/jflmEXtnT6fkGcnV6
               ↪  dGNUpHWXSpwUIDt0N88jfnEqekx4S+KDCKg99sGEeHeT65fKS8
               ↪  lB0gjHMt9A0riwIDAQAB\n"
11          }
12        }
13      }
14    ],
15    ...
```

AgreementCall

The *agreement* function is a particular function compared to the others and therefore must be managed ad-hoc. For this function you need:

1. `contractId`: identifier of the contract. This function call will create a *new instance of the indicated contract*;
2. `arguments`: the list of arguments of the function to call;
3. `parties`: is a map that provides the association between the party name in the contract and the user's *address* and *public key*. An address is a compact representation of the public key, in particular, it is the hash of the public key. For example:

```
1       ...
2       "parties": {
3         "Bob": {
4           "address":
             ↪  "f3hVW1Amltnqe3KvOT00eT7AU23FAUKdgmCluZB+nss=",
5           "publicKey": "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBg
             ↪  QDErzzgD2ZslZxciFAiX3/ot7lrkZDw4148jFZrsDZPE6CV
             ↪  s9xXFSHGgy/mFvIFLXhnChO6Nyd2be3lbgeavLMCMVUiTSt
             ↪  Xr117Km17keWpb3sItkKKsLFBOcIIU8XXowI/OhzQN2XPZY
             ↪  ESHgjdQ5vwEj2YyueiS7WKP94YWz/pswIDAQAB"
6         },
7         "Alice": {
8           "address":
             ↪  "ubL35Am7TimL5R4oMwm2OxgAYA3XT3BeeDE56oxqdLc=",
9           "publicKey": "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBg
             ↪  QCo/GjVKS+3gAA55+kko41yINdOcCLQMSBQyuTTkKHE1mhu
             ↪  /TgOpivM0wLPsSga8hQMr3+v3aR0IF/vfCRf6SdiXmWx/jf
             ↪  lmEXtnT6fkGcnV6dGNUpHWXSpwUIDt0N88jfnEqekx4S+KD
             ↪  CKg99sGEeHeT65fKS8lB0gjHMt9A0riwIDAQAB"
10        }
11      },
12      ...
```

`Alice` and `Bob` are the names of the variables representing the parties in the contract. With this function call, these variables now have an associated address and public key.

The `AgreementCall` is a bit more complicated than just `FunctionCall`. The reason is that in order to agree to a contract, both parties to the contract must sign a **single message**. There are different ways to collect signatures to add to your message. An easy way could be for the two parties to the contract to agree on the terms of a contract (i.e., the cost of a service) by means of communication channels such as chat or email. One of the two parties creates the `AgreementCall` message, signs it with his private key and sends it via chat or email to the other party. The other party downloads the message, checks that the previously agreed values have been entered, checks that the other party's signature is legitimate and also signs the message. Once this procedure has been carried out, one of the two actors sends the `AgreementCall` message *only once* to the server. Another context could be an external application that relies on a *Stipula server*. This application can perform the same operations described in the previous example, hiding all the steps through a single graphical interface. Once all the signatures of the actors have been collected, the application, based on the *Stipula server*, will send the `AgreementCall` to the *Stipula server*. The advantage of having structured the architecture and the communication in this way is that it does not place any constraints on the communication between the actors. When the *Stipula server* has received the `AgreementCall` message with legitimate signatures, the architecture will create a new instance of the contract chosen by the actors.

The `AgreementCall` request it will then be directed to the virtual machine.

GetAssetById

This message allows you to obtain information about a specific asset, given an identifier. In fact, the only required value is the *identifier of the asset* whose information is to be obtained.

This request will then be routed to *Storage*.

GetOwnershipsByAddress

This message allows you to get all spent and unspent funds from a specific address. In fact, the only required value is a `address`. The use of an address allows to transmit less data in the socket and to carry out less computations in the *Storage* to find the address associated with the public key.

This request will then be routed to *Storage*.

4.3.5 Interaction with Storage

The only requests that allow this module to interact *directly* with the *Storage* module are `GetAssetById` and `GetOwnershipsByAddress`. These messages require to be able to obtain information, that is, to perform a *read* operation from the *Storage* module. In fact, all the requests that imply a modification of a piece of information are requests that are addressed to the compiler and virtual machine modules. Only these two modules can actually *write* to *Storage*.

4.4 Compiler

A *compiler* is a software program that translates source code, written in a high-level programming language, into machine code that can be executed by a computer. In this case, we want to develop a compiler to translate the high-level language *Stipula* into a language that can be executed by a machine: the *Stipula bytecode*.

A compiler is made up of several components, which can be grouped into:

1. **Front-end**: it is the part of the compiler that deals directly with the source code and produces an internal representation that can be easily processed by the back-end. The front-end output is usually an intermediate representation such as an *abstract syntax tree*, which can be optimized and transformed by the back-end before being translated into machine code or some other target language. This involves tasks such as *lexical parsing* (splitting input into tokens), *syntax parsing* (parsing tokens into a parse tree or an abstract syntax tree), and *semantic analysis* (making sure the input conforms to the rules of the language and generating an intermediate representation);

2. **Back-end**: is responsible for generating executable code from the intermediate representation produced by the front-end. This involves several stages, including:

   (a) *Optimization*: this phase involves the analysis of the intermediate code and its transformation to produce a more efficient code;

   (b) *Code generation*: in this phase, the optimized intermediate code is transformed into executable machine code. This involves translating each intermediate code instruction into one or more machine instructions, taking into account the target hardware platform and processor specific instruction set;

   (c) *Linking*: The generated code is combined with any required runtime libraries and other resources to produce an executable program.

   A compiler's backend is typically heavily dependent on the target architecture, and different backends may be needed for different hardware platforms or operating systems.

The typical structure of a compiler includes the following components:

1. **Lexer**: this component reads the source code character by character and decomposes it into *token*. A token is a sequence of characters that represents a significant unit of the language, such as a keyword, an identifier or an operator;

2. **Parser**: this component takes the stream of tokens generated by the lexer and builds a **syntax tree** or an **abstract syntax tree** (**AST**) which represents the syntax structure of the program. The AST captures the hierarchical relationships between language constructs in the program;

3. **Semantic Analyzer**: This component checks the AST for semantic correctness, such as type checking and error detection. Ensures that the program follows the rules of the programming language and can run correctly;

4. **Intermediate code generator**: this component translates the AST into an intermediate representation, i.e. a machine-independent low-level code that can be optimized and further translated into executable code;

5. **Code optimizer**: this component applies various optimization techniques to intermediate code to improve its efficiency and reduce its size;

6. **Code generator**: this component translates the optimized intermediate code into machine code that can be executed by the target processor;

7. **Linker**: this component combines the object files produced by the code generator into a single executable file and resolves any external references between them.

An external tool (see section 4.4.1) was used to automate the development of some parts of the compiler. The part that was implemented manually is the part that concerns the *mapping* of the *Stipula* instructions into *Stipula bytecode* instructions.

For the implementation of the compiler not all the steps described have been followed:

1. The *linker* is not useful in the current state of the language;

2. The *intermediate code generator* is replaced by the *code generator*, as the *Stipula bytecode* already represents the target language;

3. The *code optimization* phase is missing, especially when it comes to analyzing and solving *syntactic sugar*. In particular, see 5.1.1 for an illustration of this problem.

4.4.1 Grammar, lexer e parser

Grammar

The original grammar of the *Stipula* language (Silvia Crafa, Cosimo Laneve and Adele Veschetti, 2022b and Silvia Crafa, Cosimo Laneve and Adele Veschetti, 2022c) is as follows:

<*prog*> ::= `'stipula'` <*id*> '`{`' <*declist*>* <*agreement*>? <*fun*>+ '`}`';

<*agreement*> ::= (`'agreement'` '`(`' <*disputer*> (`','` <*disputer*>)* '`)`' '`(`' <*vardec*> (','
<*vardec*>)* '`)`' '`{`' (<*assign*>)+ '`}`' '`==>`' '`@`' <*state*>);

<*fun*> ::= (('`@`' <*state*>)+ <*disputer*> (',' <*disputer*>)* '`:`' <*id*> '`(`' (<*vardec*> (','
<*vardec*>)*)? '`)`' '`[`' (<*assetdec*> (',' <*assetdec*>)*)? '`]`' ('`(`' <*prec*> '`)`')? '`{`' <*stat*>+ ';'
<*events*>+ '`}`' '`==>`' '`@`' <*state*>);

<*assign*> ::= (<*disputer*> (',' <*disputer*>)* '`:`' <*vardec*> (',' <*vardec*>)*);

<*stat*> ::= '`_`' | (<*value*> ('`->`' | '`-o`') <*value*> (',' <*value*>)?) | <*ifelse*>;

<*ifelse*> ::= (`'if'` '`(`' <*expr*> '`)`' '`{`' <*stat*>+ '`}`' (`'else if'` '`(`' <*expr*> '`)`' '`{`' <*stat*>+
'`}`')* (`'else'` '`{`' <*stat*>+ '`}`')?);

<*events*> ::= '`_`' | (<*expr*> '`>>`' '`@`' '`id`' '`{`' <*stat*>+ '`}`' '`==>`' '`@`' '`id`');

<*prec*> ::= <*expr*>;

<*expr*> ::= ('`-`')? <*term*> (('`+`' | '`-`' | '`||`') <*expr*>)?;

<*term*> ::= <*factor*> (('`*`' | '`/`' || '`&&`') <*term*>)?;

<*factor*> ::= <*value*> (('`==`' | '`<`' | '`>`' | '`<=`' | '`>=`' | '`!=`') <*value*>)?;

<*varasm*> ::= <*vardec*> '`=`' <*expr*>;

<*declist*> ::= <*type*> <*strings*>;

<*type*> ::= '`asset`' | '`field`' | '`int`' | '`real`' | '`boolean`' | '`party`' | '`string`' | '`time`' | '`init`';

<*state*> ::= <*strings*>;

<*disputer*> ::= <*strings*>;

<*vardec*> ::= <*strings*>;

<*assetdec*> ::= <*strings*>;

<*value*> ::= <*number*> | '`now`' | '`(`' <*expr*> '`)`' | <*strings*> | '`_`' | ('`true`' | '`false`');

<*id*> ::= '`id`';

<*strings*> ::= `SINGLE_STRING` | `DOUBLE_STRING` | '`id`';

<*real*> ::= <*number*> '`.`' <*number*>;

<*number*> ::= `INT` | `REAL`;

This grammar has a limitation regarding assets. Suppose you need to write a contract to swap two assets. The code could be as follows:

```
1    stipula SwapAsset {
2      asset assetA, assetB
3      field amountAssetA, amountAssetB
4      ...
```

However, from this code it is not possible to understand which assets are being referred to exactly. That is, if Alice wants to swap *assetA* for *assetB* owned by Bob, there is no specific indication of these assets in the code. The change that was made to the grammar is as follows:

<*declist*> ::= (<*assetdecl*>)? (<*fielddecl*>)?;

<*assetdecl*> ::= '`asset`' <*strings*> '`:`' <*strings*>;

<*fielddecl*> ::= <*type*> <*strings*>;

<*type*> ::= '`field`' | '`int`' | '`real`' | '`boolean`' | '`party`' | '`string`' | '`time`' | '`init`';

By doing so, it is possible to specify the assets that must be accepted by the contract. Thus, the previous code in *Stipula* transforms with the grammar change as follows:

```
1  stipula SwapAsset {
2    asset assetA:stipula_assetA_ed8i9wk,
       ↪  assetB:stipula_assetB_pl1n5cc
3    field amountAssetA, amountAssetB
4    ...
```

The B appendix illustrates the rules of the defined grammar previously (see 4.4.1) translated into **ANTLR**.

Lexer, Parser and ANTLR

ANTLR (*ANother Tool for Language Recognition*) is a **lexer** and **parser** generator that can be used to create compilers, interpreters and other language processing tools. It is a tool well known for its ability to generate highly efficient parsers that can handle complex and context sensitive grammars. It also provides a simple syntax for defining grammars, which makes it easier to create parsers for new languages or formats.

In order to use this tool, it is necessary to convert the grammar defined in the previous section, following the rules established by ANTLR (see the B appendix). Version 4.10 was used for this project (*ANTLR v4.10*).

The tool is written in Java and in order to use it you need to execute a `.jar` file. In particular, the command to generate the classes that implement the lexer and the parser is the following:

```
java -jar antlr-4.10-complete.jar -visitor Stipula.g4
```

In order to use the lexer and parser produced by ANTLR, it is necessary to integrate the latter tool into the project. The integration is specified in the D appendix.

4.4.2 Generation of the bytecode

This stage occurs after the parser has produced the abstract syntax tree. In particular, the AST is visited and for each instruction of the *Stipula* language one or more bytecode instructions are generated. In this phase, any syntactic sugar present in the contract is also resolved. The translation of the syntactic sugar takes place by generating fixed structures in bytecode language, that is, once the syntax variant of a specific instruction has been recognized, this is always translated into a fixed structure. This practice allows in the execution phase not to worry about the presence of any syntactic sugar to be resolved.

Once compiled, the source code of the contract and the compiled are stored in the *Storage* module.

In the next section we will introduce the *Stipula bytecode* language, that is, its functioning and its instructions.

4.5 Stipula bytecode

This language was designed to mirror the functionality of the high-level language and to run on a *stack-based* virtual machine. A summary table of the instructions is shown below 4.1: the `-` symbol means that the statement takes no value as input or returns no value as output, while the `*` means that the instruction accepts a value of any type or outputs a value of any type.

Table 4.1: Table of *Stipula bytecode* instructions.

Instruction	Behavior
PUSH	$- \rightarrow *$
HALT	$- \rightarrow -$
ADD	$(\texttt{int}, \texttt{int}) \rightarrow \texttt{int}$, $(\texttt{real}, \texttt{real}) \rightarrow \texttt{real}$, $(\texttt{asset}, \texttt{asset}) \rightarrow \texttt{real}$, $(\texttt{asset}, \texttt{real}) \rightarrow \texttt{real}$, $(\texttt{real}, \texttt{asset}) \rightarrow \texttt{real}$, $(\texttt{time}, \texttt{time}) \rightarrow \texttt{time}$
SUB	$(\texttt{int}, \texttt{int}) \rightarrow \texttt{int}$, $(\texttt{real}, \texttt{real}) \rightarrow \texttt{real}$, $(\texttt{asset}, \texttt{asset}) \rightarrow \texttt{real}$, $(\texttt{asset}, \texttt{real}) \rightarrow \texttt{real}$, $(\texttt{real}, \texttt{asset}) \rightarrow \texttt{real}$
MUL	$(\texttt{int}, \texttt{int}) \rightarrow \texttt{int}$, $(\texttt{real}, \texttt{real}) \rightarrow \texttt{real}$, $(\texttt{asset}, \texttt{asset}) \rightarrow \texttt{real}$, $(\texttt{asset}, \texttt{real}) \rightarrow \texttt{real}$, $(\texttt{real}, \texttt{asset}) \rightarrow \texttt{real}$
DIV	$(\texttt{int}, \texttt{int}) \rightarrow \texttt{int}$, $(\texttt{real}, \texttt{real}) \rightarrow \texttt{real}$, $(\texttt{asset}, \texttt{asset}) \rightarrow \texttt{real}$, $(\texttt{asset}, \texttt{real}) \rightarrow \texttt{real}$, $(\texttt{real}, \texttt{asset}) \rightarrow \texttt{real}$
INST	$- \rightarrow -$
AINST	$- \rightarrow -$
GINST	$- \rightarrow -$
LOAD	$- \rightarrow *$
ALOAD	$- \rightarrow *$
GLOAD	$- \rightarrow *$
STORE	$* \rightarrow -$
ASTORE	$* \rightarrow -$
GSTORE	$* \rightarrow -$
AND	$(\texttt{bool}, \texttt{bool}) \rightarrow \texttt{bool}$
OR	$(\texttt{bool}, \texttt{bool}) \rightarrow \texttt{bool}$
NOT	$\texttt{bool} \rightarrow \texttt{bool}$
JMP	$- \rightarrow -$
JMPIF	$\texttt{bool} \rightarrow - \mid\mid \texttt{bool}$
ISEQ	$* \rightarrow \texttt{bool}$
ISLE	$* \rightarrow \texttt{bool}$
ISLT	$* \rightarrow \texttt{bool}$
DEPOSIT	$(\texttt{asset}, \texttt{asset}) \rightarrow -$
WITHDRAW	$(\texttt{real}, \texttt{asset}, \texttt{party}) \rightarrow -$
RAISE	$- \rightarrow \texttt{str}$
TRIGGER	$\texttt{time} \rightarrow -$

4.5.1 Types

This section introduces the types that are supported by the *Stipula bytecode*.

Integers, Strings e Booleans These are the simplest types to implement, as only one field in the Java object representation is required for storing the value. Examples of declarations:

1. Integer: `int <variable_name> 123`;
2. String: `str <variable_name> abc`;
3. Boolean: `bool <variable_name> true` or `bool <variable_name> false`.

Time From an implementation point of view, this type is similar to the previous one. However, the value of a variable of type `time` represents a certain amount of time expressed in *seconds*. Example of declaration: `time <variable_name> 123`.

Real numbers This type was implemented in a simple way: there are two fields, one representing the number for *extended*, that is, without the comma; the other indicates the *number of decimals* to apply to the number contained in the first field. For example, to encode the number 134.28 you would write 13428 2, that is, 13428 represents the number in full and 2 represents the number of decimals to apply to that value. Example of declaration: `float <variable_name> 123 1` means 12.3.

Party The Java object that represent the type `party` in the bytecode language is structured as follows:

```
9   ...
10  public class Party implements Serializable {
11    private final String address;
12    private final String publicKey;
13
14    public Party(String publicKey) throws
      ↪ NoSuchAlgorithmException {
15        this.publicKey = publicKey;
16
17        // Hash the public key
18        Base64.Encoder encoder = Base64.getEncoder();
19        MessageDigest digest =
          ↪ MessageDigest.getInstance("SHA-256");
20
21        this.address = encoder.encodeToString(digest.digest(publ⌋
          ↪ icKey.getBytes(StandardCharsets.UTF_8)));
22    }
23  ...
```

1. `publicKey` represents a user's public key;
2. `address` is a more compact representation of the public key, in particular, it is the *hash* of the public key. Having this field will allow you to use less space and make *Storage* searches faster.

Example of declaration:

```
party <variable_name>
↪ ubL35Am7TimL5R4oMwm2OxgAYA3XT3BeeDE56oxqdLc=
```

Asset The structure of this type consists of:

1. The `real` part represents the quantity of assets contained in the variable;
2. The `assetId` field represents the identifier of the asset. This makes it possible to understand which asset the quantity specified by the real part belongs to.

Example of declaration: `asset <variable_name> 100 2 stipula_coin_asd345`.

4.5.2 Instructions of the bytecode language

In this section we will explain how the language instructions work. In section 3.3.3 it was explained what a stack-based virtual machine is. The virtual machine is able to manage the values on the stack by means of two operations. When an instruction takes a value as input, it means that it *pop* from the stack. While, when an instruction returns a value in output, it means that it performs the *push* operation on the stack.

PUSH This statement allows you to insert values into the *stack*. This statement takes an input value of any type and returns no output value. Possible formats for this statement are:

1. `PUSH int <value>`;
2. `PUSH bool <value>`;
3. `PUSH str <value>`;
4. `PUSH party <value>`;
5. `PUSH time <value>` and `PUSH time now`: `now` is a reserved word and when this instruction is read by the virtual machine, this word is interpreted as the intention to get the current *timestamp*;
6. `PUSH real <value> <decimals>` (i.e., `PUSH real 13428 2`);
7. `PUSH asset <value> <decimals> <asset-id>`
   (i.e., `PUSH asset 13428 2 stipulation_coin_asd345`).

HALT This statement notifies the virtual machine that the function code is finished. If this statement is executed, it means that the entire execution of the function did not generate any errors. With this instruction, the virtual machine proceeds to store the results produced, send any payments to addresses and provide a response for the client. This statement takes no value as input and returns no output.

ADD This statement implements the *sum* mathematical operation. This statement takes two values as input and returns one value as output. In most cases, the types of the two input values must be equal to each other, and the type of the output value must be equal to the type of the input values. For example, if two values of type `int` are received as input, then the output will be of type `int`.

However, there is an exception regarding the `asset` type. In fact, manipulation of variables of this type must be done with great caution. If you want to somehow add the amount of assets owned by a specific instance of a contract with other values, this operation must absolutely not affect the amount of assets present, that is, a sum operation involving a variable of type `|asset|`,must ensure that after this operation no quantity has been lost or some quantity of asset has been generated out of thin air. The only operations that can manipulate the quantity of assets contained in the appropriate variable are the *deposit* and *withdrawal* operations, which will be described later. Therefore, the application of mathematical operations to variables of type `asset` has been limited as follows:

1. A variable of type `real` is accepted as input and one of type `asset` and the output of the operation will be a value of type `real`;
2. A variable of type `asset` is accepted as input and one of type `real` and the output of the operation will be a value of type `real`;

By doing so, it is not possible to generate or destroy quantities of assets through the use of mathematical operations.

Finally, this is the only mathematical instruction that allows you to manipulate `time` values. Therefore, this sequence of instructions:

```
1    ...
2    GLOAD waitTime
3    PUSH time now
4    ADD
5    ...
```

will add the value contained in the global variable `waitTime` at the *timestamp* calculated at the instant in which the virtual machine will read the instruction on line 3.

SUB, MUL and DIV These instructions implement the mathematical operations of *subtraction*, *multiplication* and *division*, respectively. The behavior of these instructions is similar to the operation of the add instruction, except that they cannot handle values of type `time`.

INST, AINST and GINST These statements take no value as input and return no value as output. These statements allow you to *instantiate* new variables. In particular:

1. `INST` allows to instantiate a variable in the space dedicated to the function;
2. `AINST` allows you to instantiate a variable in the space dedicated to the function's arguments;
3. `GINST` allows you to instantiate a variable in the space dedicated to global variables of the contract instance.

The possible formats for these instructions are:

1. `INST` AINST | GINST <int | bool | str | party | time> <value>|;
2. `INST` AINST | GINST time <value> | now|;
3. `INST` AINST | GINST real <value> <decimals>|;
4. `INST` AINST | GINST asset <value> <decimals> <asset-id>|;
5. `INST` AINST | GINST * <value>|: this format allows you to instantiate a variable whose type must be determined at runtime.

Before instantiating a new variable, the virtual machine will make sure that another variable with the same name does not exist in memory.

LOAD, ALOAD and GLOAD These instructions allow you to *load* a variable from memory onto the stack. These instructions take no value as input and return the variable loaded from memory as output. In particular:

1. `LOAD`: allows you to load a variable from the space dedicated to the function;
2. `ALOAD`: allows you to load a variable from the space dedicated to the function arguments;
3. `GLOAD`: allows you to load a variable from the space dedicated to global variables of the contract instance.

The instruction formats are:

```
LOAD | ALOAD | GLOAD <variable_name>
```

Before loading a variable, the virtual machine will make sure that the variable exists in memory.

STORE, ASTORE and GSTORE These statements allow you to *store* a variable in memory. These statements take a value of any type as input and return no value as output. In particular:

1. `STORE`: allows you to store a variable in the space dedicated to the function;
2. `ASTORE`: allows you to store a variable in the space dedicated to the function arguments;
3. `GSTORE`: allows you to store a variable in the space dedicated to the global variables of the contract instance.

The instruction formats are:

```
STORE | ASTORE | GSTORE <variable_name>
```

Before storing the variable, the virtual machine will make sure that the variable exists in memory.

AND and OR These statements take two *boolean* values as input and return a boolean value as output. For the `AND` statement, if both input values are `true`, then the output will be `true`, otherwise the output will be `false`. While, for the `OR` statement, if one of the two input values is `true`, then the output will be `true`, otherwise the output will be `false`.

NOT This statement takes a *boolean* value as input and returns a boolean value as output. The output of this statement is the inverse of the input value, that is, if the input is `true`, then the output will be `false`, and vice versa.

ISEQ Given two boolean inputs, this statement checks whether two values are *equal*. If the input values are equal, the output will be `true`, otherwise the output will be `false`. This statement can be applied to any type, as long as the input types are the same. The only allowed exceptions are the input pairs (`asset`, `real`) and (`real`, `asset`).

Combining this statement with the `NOT` statement it is possible to check if the two input values are different from each other: if the values are *different*, the output will be `true`, otherwise the output will be `false`.

ISLE Given two boolean inputs, this statement checks whether the first value is *less than or equal* to the second value. If this condition is true, then the output will be `true`, otherwise the output will be `false`. This statement only applies to types `int`, `real` and `assets`.

Combining this statement with the `NOT` statement it is possible to check if the first value is *strictly greater* than the second value: if this condition is true, the output will be `true`, otherwise the output will be `false`.

ISLT Given two boolean inputs, this statement checks whether the first value is *strictly less* than the second value. If this condition is true, then the output will be `true`, otherwise the output will be `false`. This statement only applies to types `int`, `real` and `assets`.

Combining this statement with the `NOT` statement it is possible to check if the first value is *greater than or equal* to the second value: if this condition is true, the output will be `true`, otherwise the output will be `false`.

JMP This statement takes no value as input and returns no value as output. This instruction represents the *unconditional jump* and allows interrupting the normal flow of execution of the function to reach a specific area of code (always within the function), indicated by a *label*. The format of the statement is `JMP <label>`. Since the high-level language *Stipula* is not a Turing complete language, it is not possible to create loops with this instruction, since the jump can only be made in *"forward"*, i.e., the search of the *label* starts from the position where the last statement was just executed and goes forward, it is not allowed to search for the label starting from the beginning of the function.

JMPIF This instruction takes as input a value of type `bool`. This instruction allows to implement the *conditional jump*, that is, it is possible to interrupt the normal execution flow and reach a specific area of code, within the function, only if a previous condition has been satisfied. The format of the statement is `JMPIF <label>`. If the

input value is `true`, then jumping to the code area indicated by `<label>` will happen and no value will be output, otherwise if the input value is `false`, the input value will be output.

DEPOSIT This statement takes two values as input, both of type `asset`, and returns no value as output. This statement allows you to *deposit* assets into an instance of a contract. The first value represents the amount of assets deposited by the user and this amount is *accumulated* into the second value, which represents the amount of assets present in the contract instance. This is one of two instructions that is allowed to directly manage assets and store updates to these variables.

WITHDRAW This instruction allows you to make payments to a specific participant of the contract. This statement takes three values as input and returns no value as output. The input values are:

1. `real`: this value represents the quantity of assets that must be withdrawn;
2. `asset`: this value represents the quantity of assets contained in the contract instance;
3. `party`: this value represents the party of the contract to which the payment must be made.

This instruction is allowed to directly manage the assets and to store the updates regarding these variables.

RAISE This statement takes no value as input and outputs a value of type `str` which will be placed on the *error stack* of the virtual machine. This statement is used to block the flow of function execution to notify an error. In this version of the architecture, the only error that can be reported to the error stack is `AMOUNT_NOT_EQUAL` and is notified when the user wants to deposit an amount of assets that does not match the amount of assets agreed at the beginning of the contract.

TRIGGER This instruction takes as input a value of type `time` and returns no output. This instruction allows you to create an event to schedule, at a precise moment, the execution of a *obligation*. The format of the instruction is `TRIGGER <obligation_function_name>`, having to for `<obligation_function_name>` we refer to the name of the function that encodes the obligation that will have to be performed.

4.5.3 Function types

In the bytecode language we can define three types of functions:

1. **agreement** function: this is the function that allows you to create a new instance of a contract. Here is an example:

   ```
   fn agreement Alice,Bob Inactive real,str
   ```

   In particular:

   (a) `fn agreement`: it specifies that the function to be performed will be a *agreement* function;

(b) `Alice,Bob`: the participants in the contract are defined. When this function is called, the addresses and public keys of the users who want to execute the contract will need to be provided. Therefore, two parameters of type `party` will have to be supplied;

(c) `Inactive`: indicates the state in which the instance of the contract must go, once the virtual machine has finished executing the *agreement* function without errors;

(d) `real,str`: it indicates that, in addition to the addresses and public keys of the users, two parameters must be supplied, one of type `real` and the other of type `str`;

2. Function representing a **obligation**: this is the function that is invoked at a given moment by an *event*. This event was previously scheduled by another function. Here is an example:

```
obligation Swap obligation_1 End
```

In particular:

(a) `obligation`: it specifies that the function to be performed is a *obligation*;

(b) `Swap`: indicates the state in which the instance of the contract must be in order to be able to execute the obligation;

(c) `obligation_1`: is the name of the function that represents the obligation;

(d) `End`: indicates the state in which the instance of the contract must go, once the virtual machine has finished executing, without errors, the function that represents the obligation;

Functions that represent obligations do not accept any parameters.

3. **generic** function: this is a generic function that can be called by a user via the `FunctionCall` message. Here is an example:

```
fn Inactive Alice deposit Swap int,asset
```

In particular:

(a) `fn`: you specify that you are going to execute a function;

(b) `Inactive`: indicates the state in which the instance of the contract must be in order to execute the function;

(c) `Alice`: indicates which participant of the contract can call this function;

(d) `deposit`: is the name of the function;

(e) `Swap`: indicates the state in which the instance of the contract must go, once the virtual machine has finished executing the function, without errors;

(f) `int,asset`: it indicates that two parameters must be supplied, one of type `int` and the other of type `asset`.

These functions differ only in their definition, the body of each type of function is no different from the other, they all use the same set of instructions and types.

In a *Stipula* contract there must be only *one agreement* function, while for the other functions it is also possible to **overloading**. Indeed, taking as an example

```
fn Inactive Alice deposit Swap int,asset
```

the following definitions are all legitimate in a *Stipula* contract:

1. `fn Inactive Alice deposit End int,asset`
2. `fn Start Alice deposit Swap int,asset`
3. `fn Inactive Bob deposit Swap int,asset`
4. `fn Inactive Alice deposit Swap asset,int`
5. `fn Inactive Alice deposit Swap int`

Those defined are just some examples of permitted overloading.

4.6 Virtual Machine

This module handles client function calls, executes contract code, and makes payments to users (*Pay-to-Party*). In addition, this module also takes care of updating contract instance information and verifying user payments (*Pay-to-Contract*).

At a certain level of abstraction it is possible to consider the virtual machine as a single component. However, the implemented implementation foresees two distinct virtual machines, which execute different programs: a virtual machine for the execution of *contracts* (**Legal Contract Virtual Machine**, 4.6.2) and a virtual machine which validates the programs written in *Script* (**Script Virtual Machine**, 4.6.3).

4.6.1 Requests queue

This component implements a *queue* that collects all requests made by clients and events for the execution of obligations. In the *Stipula* language, for a contract, if at the same time t a request from the client to perform a function and a request to perform an obligation arrive simultaneously, the execution of the obligation takes **precedence** over the execution of the function requested by the client. To do this, the `RequestQueue` manages two queues: one queue collects client requests (`functionCallRequests`) and one queue collects obligation execution requests (`obligationRequests`). The `RequestQueue` object has two main methods:

1. `enqueue`: this method allows you to add a request to the queue;
2. `dequeue`: this method allows you to get a request from one of the two queues. If the `obligationRequests` has items, this method will return an item from this queue. If the `obligationRequests` is empty, then an item from the `functionCallRequests` queue will be returned.

Access to these queues is controlled by a *mutex*, in order to properly handle precedence between requests. The approach used in the current implementation of the architecture may be a performance limitation. In the next chapter some optimizations have been proposed (see 5.2).

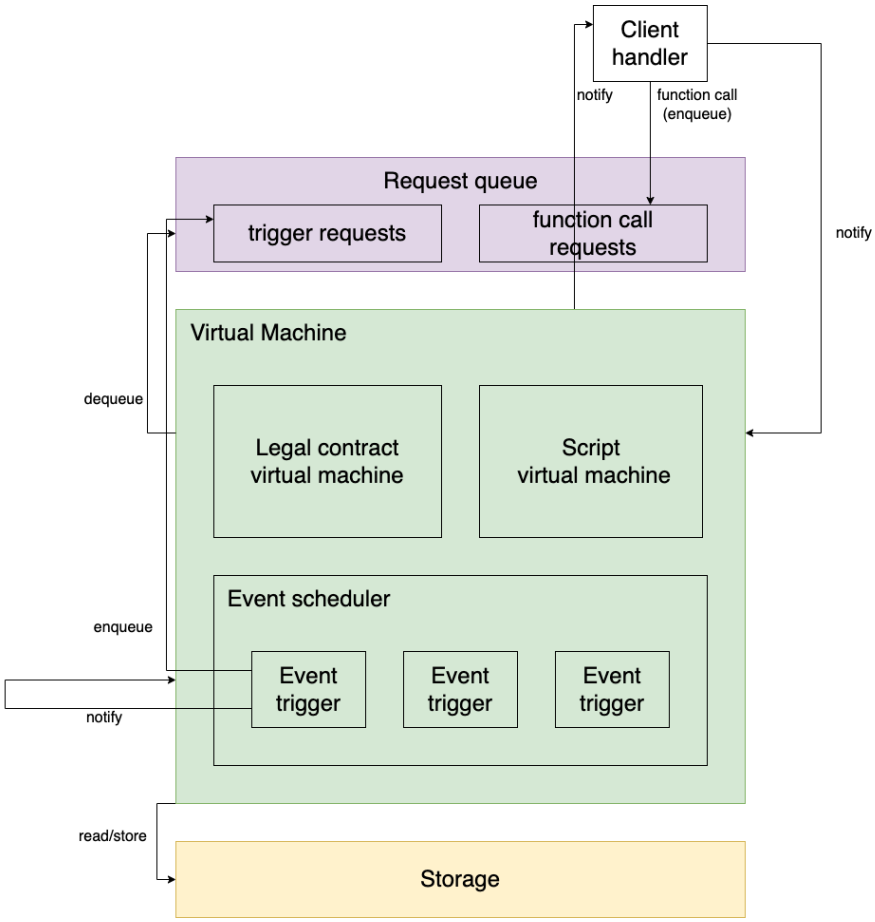


Figure 4.2: Virtual machine.

4.6.2 Legal Contract Virtual Machine

This component performs contract functions written in *Stipula bytecode*. The instructions of this virtual machine are those listed in the 4.1 table. The functioning of this component is very simple: given as input a function of a contract written in bytecode and the arguments of this function, the virtual machine sequentially executes each instruction. If one or more errors are thrown during the execution of the function, the virtual machine interrupts the execution and returns the errors in a special *error stack*; otherwise, execution proceeds until the `HALT` instruction is reached, which corresponds to the end of the function.

In this virtual machine there are several *memory zones*:

1. `stack`: this is the memory area used by the virtual machine to manipulate the values by means of the instructions read;
2. `scopeSpace`: this space is dedicated to the storage of local variables to the function;
3. `argumentsSpace`: this space is dedicated to storing the arguments of the function;
4. `globalSpace`: this space is dedicated to storing global variables of the contract instance. This space is valued through the information saved in the *Storage* module;
5. `singleUseSealsToCreate`: this space is dedicated to the temporary storage of the *single-use-seals* to be created. When a function whose code expects to perform one or more *Pay-to-Party* is executed, the execution is not momentarily interrupted to send the payments. We want to ensure atomicity in the execution of the code of a function. Therefore, when the virtual machine realizes that it needs to make a payment to one or more users, it temporarily stores the single-use-seals it has to create. Once the virtual machine finishes executing the function and the execution has not generated any errors, then we will proceed to perform the different *Pay-to-Party*;
6. `createEventRequests`: similarly to the previous point, when the virtual machine reads the `TRIGGER <obligation_function_name>` instruction, it stores in this dedicated space all the events it will have to create once the execution of the function has finished.

In addition, there are two other important fields:

1. `executionPointer`: this field indicates the current instruction that has been executed;
2. `offset`: a full contract is never input to the virtual machine. Only the code of the function to be executed is loaded, therefore the initial value of the `executionPointer` will always be zero. However, when debugging a contract it is useful to have a reference to the line of code that threw an error against the full code of the contract, and not the local code of the function. For this reason, this field stores the line number where the function code starts in the contract and when an error is thrown, in the logs it is possible to have both the line number local to the function and the global line number of the complete contract . Thus, the line number that takes into account the position it is in the contract is given by `offset` + `executionPointer`.

4.6.3 Script Virtual Machine

Single-use-seal and Ownership

In the previous chapter, the concept of *single-use-seal* (see 3.5.2) was introduced as a model for asset management. The structure of a single-use-seal was introduced earlier (see 4.3.4). To ensure that a single-use-seal can only be spent by the rightful owner, this seal is *blocked* using a specific program written in *Script* language. This program is saved in the `lockScript` field and is stored along with the other single-use-seal information. If a user wants to spend a specific single-use-seal, he must provide proof to prove rightful ownership of the funds. The proof is coded as another program written in *Script*, which allows you to *unlock* the `unlockScript` program. When a user provides this program as proof of ownership of the single-use-seal, he is demonstrating the *ownership* of the funds. In fact, when a user wants to make a *Pay-to-Contract*, the user provides the proof in the `FunctionCall` message (see section 4.3.4). The proof is coded in the Java object `Ownership` and is structured as follows:

1. `String contractInstanceId`: it indicates to which instance of the contract the payment must be made;
2. `SingleUseSeal singleUseSeal`: indicate the funds to be spent;
3. `String unlockScript`: the program that allows you to unlock the `lockScript` contained in the `singleUseSeal` object.

Joining `unlockScript` and `lockScript` it is possible to check if the user is the actual owner of the single-use-seal he wants to spend. The main idea of this mechanism was formulated in Bitcoin in 2009 and is called **Pay-to-Public-Key-Hash** (**P2PKH**) (Antonopoulos, 2017). This was one of the very first mechanisms to be able to make payments in the Bitcoin network. The `lockScript` program can only be unlocked if in the `unlockScript` program cryptographic proof is provided via the funds holder's private key. In this way, when a user wants to pay for an instance of a contract, it is the user himself who voluntarily transfers the *ownership* of a single-use-seal to the instance of the contract.

Script

In the previous chapter, the *Script* language was introduced (see 3.5.2). The instructions of this language are very limited and most of them are separate from the instructions of the *Legal Contract Virtual Machine*. The instructions of the *Script Virtual Machine* are listed in the 4.2 table and it is possible to notice the difference in the sets of instructions between the two virtual machines in the image 4.3 . Furthermore, this language only allows you to handle values that are of type `bool` or `str` (see image 4.4).

Table 4.2: Table of *Script Virtual Machine* instructions.

Instruction	Behavior
PUSH	$- \rightarrow *$
HALT	$- \rightarrow -$
DUP	$* \rightarrow (*, *)$
SHA256	$\texttt{str} \rightarrow \texttt{str}$
EQUAL	$(\texttt{str}, \texttt{str}) \rightarrow -\|\|\texttt{str}$
CHECKSIG	$(\texttt{str}, \texttt{str}) \rightarrow \texttt{bool}$

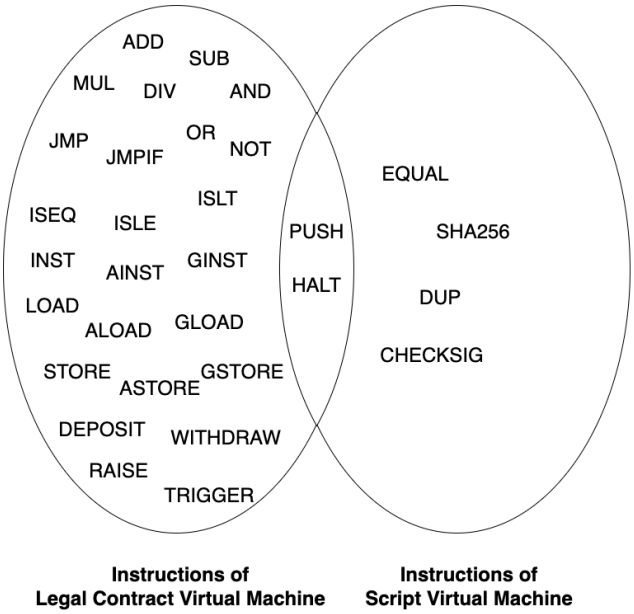


Figure 4.3: The instruction sets of the two virtual machines.

PUSH and HALT These statements have the same behavior as those defined for the *Legal Contract Virtual Machine*. The only difference is that these statements operate only on values of type `bool` and `str`.

DUP This statement takes a value of any type as input and outputs two values that have the same type as the input value. This *duplicates* the value received as input, that is, it *pops* from the stack and performs two *pushes* of the same value.

SHA256 This instruction takes as input a value of type `str` and outputs a value of the same type. This instruction calculates the SHA256 hash of the input value.

EQUAL Given two inputs of type `str`, this statement checks whether the two strings are *equal*. If the two values are equal, no value is returned, if instead the values are not equal, then a value of type `str` in the *error stack* of the virtual machine.

CHECKSIG This instruction takes as input two values of type `str` and outputs a value of type `bool`. The first value represents a *public key*, while the second represents a *signature*. This instruction allows you to check if, using the public key received as input, the signature is valid or not.

Instruction table A summary table of the instructions is shown below: the `-` symbol means that the statement takes no value as input or returns no value as output, while the `*` means that the instruction accepts a value or outputs a value of type `bool` or `str`.

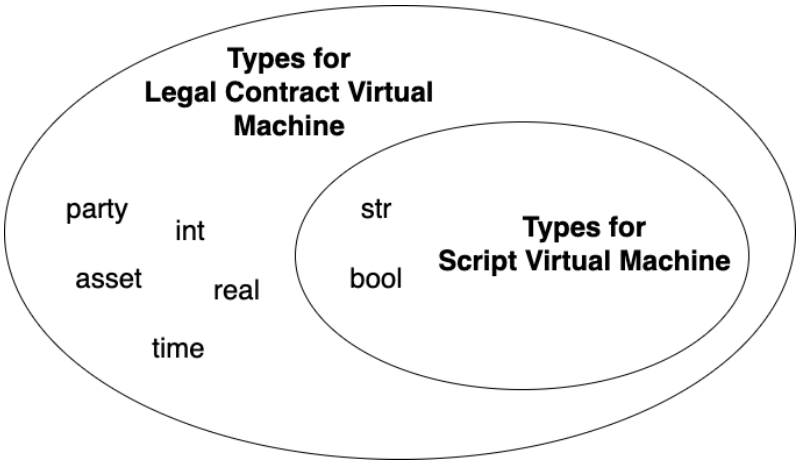


Figure 4.4: Sets of the types of the two virtual machines.

LockScript and UnlockScript

Having illustrated the *Script* and *P2PKH* language, give the well-defined structure of `lockScript` and `unlockScript`:

1. `lockScript`: `DUP SHA256 PUSH str <pub_key_hash> EQUAL CHECKSIG`;
2. `unlockScript`: `PUSH str <signature> PUSH str <pub_key>`;

where,

1. `<pub_key>`: is the public key of the user in possession of the single-use-seal;
2. `<pub_key_hash>`: corresponds to the SHA256 hash of the public key;
3. `<signature>`: corresponds to the signature of the identifier of the single-use-seal to be spent.

The `<signature>` corresponds to the cryptographic proof that only the user can provide to demonstrate possession of the single-use-seal. Signing the single-use-seal identifier provides *unique* cryptographic proof and cannot be reused to prove ownership of other funds. So, when the user sends the signature and his public key to a *Stipula* server or node, anyone can check it. Therefore, if the signature were made using information that can be *reused* to prove possession of multiple funds, this would lead to a major security problem, as anyone can verify the signature and reuse the information used in the signature to misappropriate other funds.

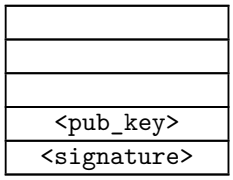
The union of `lockScript` and `unlockScript`, create the following program which will be validated by the virtual machine:

```
1  PUSH str <signature> PUSH str <pub_key> DUP SHA256 PUSH str
   ↪  <pub_key_hash> EQUAL CHECKSIG
```

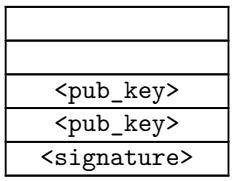
The program is evaluated as follows (an example is illustrated by observing the evolution of the stack):

1. `PUSH str <signature>`: la `<signature>` is loaded onto the stack

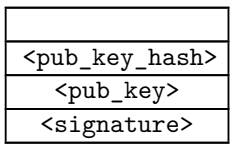
2. `PUSH str <pub_key>`: the public key is loaded onto the stack



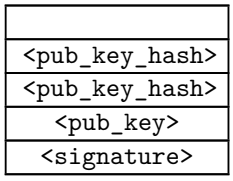
3. `DUP`: you duplicate the last element of the stack, which in this case is the public key



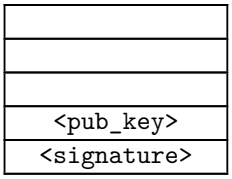
4. `SHA256`: the hash of the last element of the stack is computed, which in this case is the previously duplicated public key



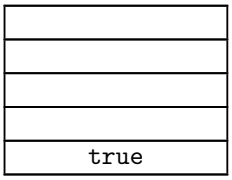
5. `PUSH str <pub_key_hash>`: the public key hash is loaded onto the stack. This hash is taken from the `unlockScript`



6. `EQUAL`: occurs if the computed hash is the hash of the `unlockScript` it is equal or less



7. `CHECKSIG`: occurs if the `<signature>` is valid with the `<pub_key>` present in the stack. If the check is successful, `true` will be pushed onto the stack, otherwise `false`



The example just illustrated described all the operations that are performed by the *Script Virtual Machine*.

4.6.4 Description of the execution flow of a function of a contract

This section illustrates the execution flow of a generic contract function (see figure 4.5). More precisely, let's suppose that the generic function requires as input a value of type `asset`, and that therefore, the user has to make a payment. The flow is as follows:

1. A `FunctionCall` message is received (see section 4.3.4): the `ClientHandler` performs all checks on the format of the message and the signature. After that, the `ClientHandler` adds this new request to the *queue of requests* (**1.1**, in figure 4.5) and *notifies* the virtual machine (**1.2**). The notification action of the virtual machine is useful in case the latter is waiting for new requests, but the request queue is empty;

2. Suppose that the only request in the request queue is the one added in the previous point. The virtual machine dequeues the only request present (**2**);

3. As mentioned previously, this is a function call that requires an asset as a parameter, therefore, in the `FunctionCall` there is all the information to verify if the funds sent are actually in the user's possession and if they are of the requested quantity. The verification of possession of the sent *single-use-seal* is delegated to the *Script Virtual Machine* (**3**). If the checks fail, the virtual machine notifies the `ClientHandler` (**6**);

4. If the verification of the *script* gives a positive result, then we proceed to execute the function indicated by the request. The execution of the function is delegated to the *Legal Contract Virtual Machine* (**4**);

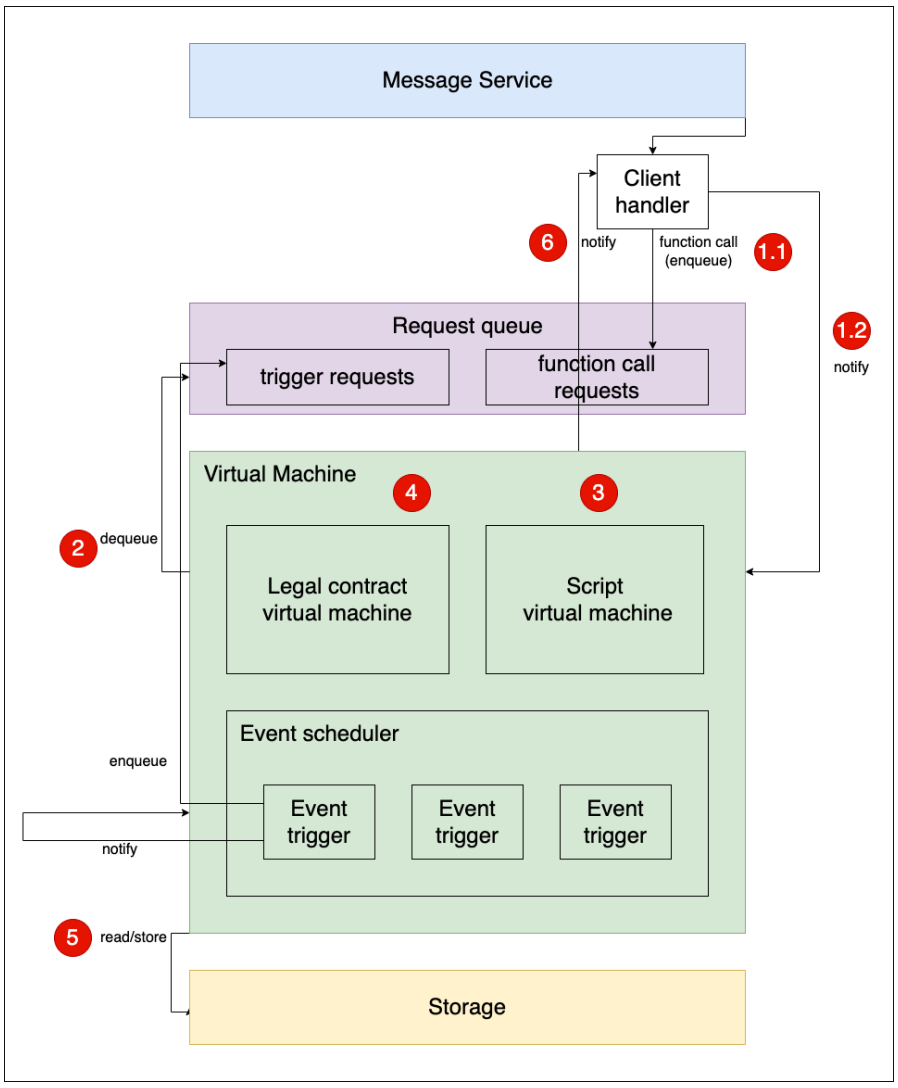


Figure 4.5: Flow of the execution of a function of a function of a contract.

5. When the execution of the function ends and there are no errors, all the modifications concerning the global variables, the change of the state of the contract and the updating of the single-use-seal, which now can no longer be spent in other contract instances are sent to the *Storage* module (**5**);

6. Finally, the virtual machine notifies the `ClientHandler`, returning a response regarding the success or failure of the function execution (**6**).

Next, concrete examples of some contract examples will be shown (see section 4.8).

4.6.5 Pay-to-Party

Previously, we discussed *Pay-to-Contract*, that is, how a user makes a payment to an instance of a contract. When, on the other hand, it is the instance of a contract that has to send payments to one or more users, this method is called *Pay-to-Party*. Again, this concept was introduced in the previous chapter (see 3.5.2). This mechanism is much simpler than *Pay-to-Contract*. When a certain function of a contract expects to send a payment to a user, the virtual machine performs all the preliminary checks, for example, it makes sure that it does not disappear by the amount of assets from the funds present in the contract instance. Once these checks have been made, the virtual machine creates new *single-use-seals*, locking them with the public key of the recipient of the funds. Specifically, the `lockScript` will have the following structure: `DUP SHA256 PUSH str <pub_key_hash> EQUAL CHECKSIG`, where `<pub_key_hash>` corresponds to the SHA256 hash of the payment recipient's public key; By doing so, these funds are now no longer owned by the contract instance, but by a specific user. As explained above (see 4.6.3), only the new owner of the funds will be able to spend them.

4.6.6 Obligations

In the context of the *Stipula* language, *obligations* are formulated into commitments that are verified at a given time and issue a corresponding penalty if the obligation has not been fulfilled. From an implementation point of view, an obligation consists in the *scheduling* of a *event* which at a given moment will call a specific function of the contract. If the obligation has been fulfilled, then there won't be the conditions to be able to execute the function, otherwise the virtual machine will execute the function, applying penalties.

Scheduling of an event and description of the flow of execution of an obligation

Scheduling always occurs through the execution, by a function, of a piece of code similar to the following:

```
1    ...
2    GLOAD waitTime
3    PUSH time now
4    ADD
5    TRIGGER obligation_1
6    ...
```

where, from line 2 to line 4 we define the time t in which the obligation must be performed (if the conditions allow it), and in line 5 we specify which function must be performed at time t. When the machine finishes executing the function, a `CreateEventRequest` object is created, in which the name of the function to be called and the time t in which this function must be called must be present. After that, this object is incorporated into the `EventSchedulingRequest` object, which also contains information about the contract instance. This last object is added to a list of `EventTrigger`, managed by `EventScheduler`, which collects all the scheduled events. `EventTrigger` is an object that extends the `TimerTask` class, which allows you to create a thread and carry out tasks at a set time t. From here we illustrate the execution flow (see figure 4.6):

1. When the time t is reached, the `EventTrigger` adds the `EventSchedulingRequest` object to the request queue (**1.1**). This way, the next request that the virtual machine executes will be a request to perform an obligation. After that, `EventTrigger` notifies the virtual machine if it is waiting for new requests, but the request queue is empty (**1.2**);

2. Suppose that the only request in the request queue is the one added in the previous point. The virtual machine dequeues the only request present (**2**);

3. If the conditions are satisfied, the virtual machine proceeds to execute the function that represents the obligation, and therefore, to apply the penalties; otherwise the virtual machine does not perform the function. The condition for being able to perform an obligation is if the current state of the contract instance coincides with the state in which the obligation must be performed (**3**);

4. When the execution of the function ends and there are no errors, all the modifications concerning the global variables, the change of the state of the contract and the updating of the single-use-seal, which now can no longer be spent in other contract instances are sent to the *Storage* module (**4**);

Next, concrete examples of some contract examples will be shown (see section 4.8).

4.7 Storage

This module allows you to store all the information regarding contracts, contract instances and their evolution, assets and all asset transfers between users and contract instances.

4.7.1 LevelDB

LevelDB (*LevelDB Official Repository*) is an open source *key-value storage library* developed by Google. It is a light, fast and efficient storage system capable of handling large amounts of data. LevelDB is designed to provide an ordered key-value store with high performance for read and write operations. Keys and values can be of any length, and the data is sorted by key in a natural order. The data is stored as a *binary blob* and the key can be any stream of bytes. However, it is important to note that LevelDB is an unstructured database, which means it doesn't enforce a particular schema or data model, and it is up to the application developer to define how to organize and access the data. For simplicity in the development of the architecture, it was decided

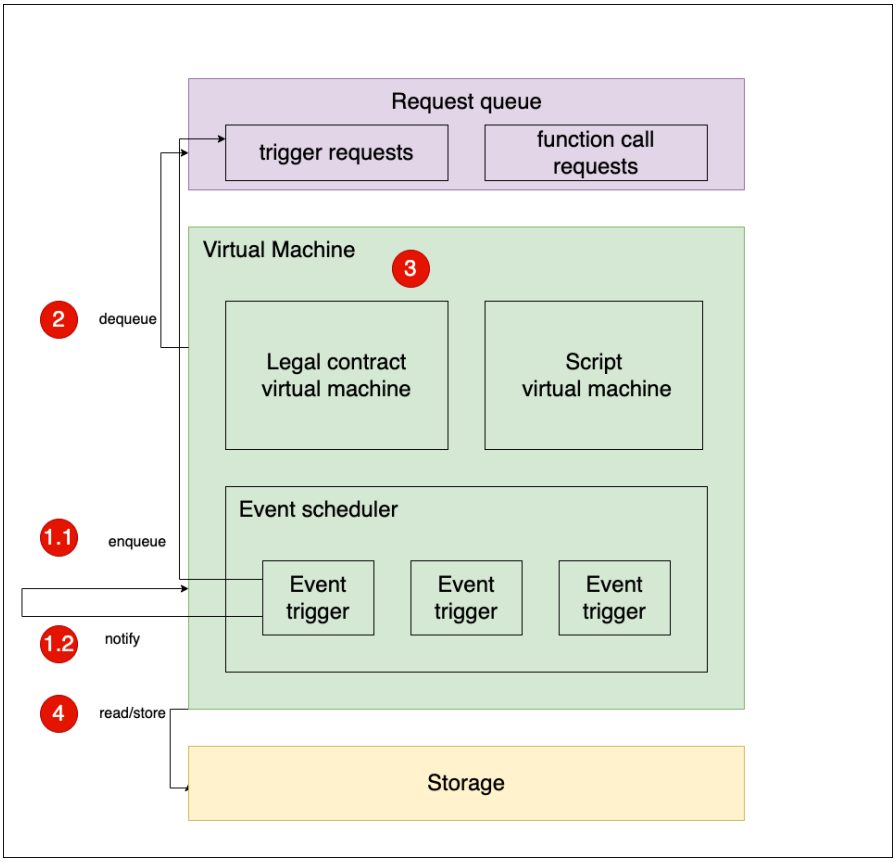


Figure 4.6: Flow of execution of an obligation.

to archive the Java objects directly, without designing a particular structure, if not following the key-value structure offered by the library.

LevelDB supports various operations, including basic *CRUD* operations (*create*, *read*, *update*, *delete*), *batch* operations, and textitsnapshot.

LevelDB is a library written in C++, but it also has *bindings* for other languages, such as Java, Python and Go. This library is used in various applications, including the Bitcoin and Ethereum blockchains.

4.7.2 Structure

The *Storage* module consists mainly of four components (see figure 4.7):

1. *Asset storage*: all the data concerning the definition of the assets are stored in this component;
2. *Ownerships storage*: this component stores all spent and unspent *single-use-seals*;
3. *Contracts storage*: this component has the task of storing all the information concerning the contract, such as the source code, the bytecode and the information for instantiating a state machine;
4. *Contract instances storage*: this component stores all the information that allows you to track the evolution of the state of a contract instance.

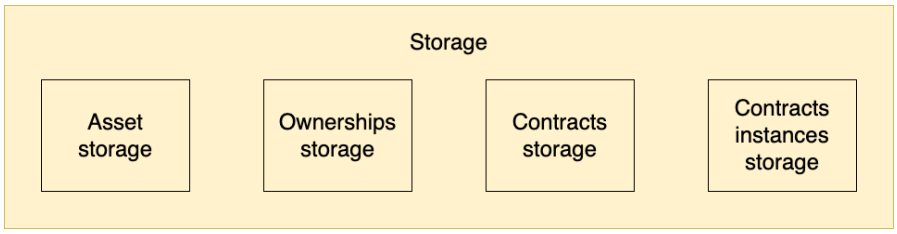


Figure 4.7: Structure of the *Storage* module.

Storage serializer There are two operations that unite all the components that allow information to be stored and they are:

1. `byte[] serialize(T data)`: this method allows you to serialize the data received as input, that is, transforming the input data into a stream of output bytes;
2. `T deserialize(byte[] bytes)`: this method allows you to deserialize the byte stream received as input into a target object `T`.

Each component of this module extends this class.

Asset storage In this class there is a main method, `getAsset`, which allows to obtain all the information concerning a specific asset, given an asset identifier as input. There is another method, `seed`, which allows you to initialize a certain number of assets when starting the *Stipula* instance. This is a method that will be removed in the

future: the need for this method to exist is closely related to the current limitations of the architecture. See section 4.9.3, to see how database *seeding* can be done, and section 5.1.3 to see a possible solution to this limitation.

Ownerships storage Also in this class there is the `seed` method, which allows you to create, in a hard-coded way, *single-use-seals* for some users. The reason is the same as the one expressed previously, that is, the need for this method is due to the current limitations of the implemented architecture.

The other methods in this class are:

1. `getFunds`: this method allows you to get all the funds, given a specific input address;
2. `getFund`: this method allows you to obtain the information of a specific *ownership*, given the property identifier and an address;
3. `addFunds`: with this method it is possible to add *ownership* to different addresses;
4. `makeOwnershipSpent`: this method allows you to update a specific *ownership* as *spent*. This method requires as input:
   (a) The *address* to which the *ownership* is associated;
   (b) The identifier of the *ownership* to update;
   (c) The identifier of the instance of the contract: this information is important as it is useful to trace from which instance of the contract the payment was made;
   (d) `unlockScript`: once the virtual machine has validated the *script* which allows to certify the user's possession of the *ownership*, the missing part of the script is saved, i.e. `unlockScript`. By doing so, this *ownership* can now no longer be spent.

Contracts storage This class contains the following methods:

1. `getContract`: this method allows you to obtain information about a specific contract, given the identifier of an input contract;
2. `saveContract`: this method allows you to store a new contract. If this method is called, the contract has been compiled successfully.

Once a contract is stored in this form, it can no longer be deleted or modified.

Contract instances storage This class contains the following methods:

1. `getContractInstance`: this method allows to obtain the information of a specific instance of a contract, given the identifier of an instance of an input contract;
2. `saveContractInstance`: this method allows you to create a new instance of a contract. If this method is called, it means that the *agreement* phase has been successful;
3. `storeGlobalSpace`: this method allows you to update the *global variables* of a specific instance of a contract. If this method is called, it means that the function execution was successful;

4. `storeStateMachine`: this method allows you to update the *current state* of the state machine of a contract instance. If this method is called, it means that the function execution was successful.

4.8 Examples

In this section, concrete examples of code will be introduced to illustrate how the implemented implementation works. Examples will include writing the contract in *Stipula*, loading and compiling the contract, and running an instance of the contract.

4.8.1 Asset swap

In this example, there are two actors, Alice and Bob, who want to trade two assets. For simplicity, the price variation that these assets may have over time is not taken into consideration, the exchange rate of these two assets is fixed by the parties to the contract when a new instance of the contract is made.

For this example there are two versions: in the first version, when Bob deposits his asset, the swap happens immediately; in the second version, when both parties deposit their assets, the swap is delegated to a *obligation*, which will be triggered after a certain time indicated by the variable `waitTimeBeforeSwapping`.

The complete code is present in the appendix A.1.1.

Agreement

This first part of the contract defines the variables for:

1. The assets: the identifiers of the assets to be exchanged in this contract are specified (line 2);
2. The quantities of assets to be traded (line 3);
3. The initial state of the contract state machine (line 4).

```
1   stipula SwapAsset {
2     asset assetA:stipula_assetA_ed8i9wk,
      ↪  assetB:stipula_assetB_pl1n5cc
3     field amountAssetA, amountAssetB
4     init Inactive
```

When two parties decide to exchange two specific assets, they make a *agreement*. In the code of this function it is possible to notice that the participants of the contract are defined and the values for `amountAssetA` and `amountAssetB`, that is, indicate the amount of assets that will have to be exchanged. Once this function has been called it means that both parties to the contract are in agreement to trade those particular assets, at an agreed rate.

```
6   agreement (Alice, Bob)(amountAssetA, amountAssetB) {
7         Alice, Bob: amountAssetA, amountAssetB
8     } ==> @Inactive
```

The following bytecode is associated with this function in *Stipula*:

```
1   fn agreement Alice,Bob Inactive real,real
2   global:
3   GINST party Alice
4   GINST party Bob
5   GINST asset assetA 2 stipula_assetA_ed8i9wk
6   GINST asset assetB 2 stipula_assetB_pl1n5cc
7   GINST real amountAssetA 2
8   GINST real amountAssetB 2
9   args:
10  PUSH party :Alice
11  GSTORE Alice
12  PUSH party :Bob
13  GSTORE Bob
14  PUSH real :amountAssetA
15  GSTORE amountAssetA
16  PUSH real :amountAssetB
17  GSTORE amountAssetB
18  start:
19  end:
20  HALT
```

In line 1 it is possible to note the signature of the function, where the name of the function (`agreement`), the participants of the contract (`Alice,Bob`), the state in which the instance of the contract will go once the execution of the function will have terminated without errors (`Inactive`) and the types of the parameters of the function (`real,real`). In this case, the function takes two parameters and both must be of type `real`.

From line 2 to line 8, the global variables of the contract are created, i.e. the participants of the contract (lines 3-4), the variables that will contain the assets that will have to be exchanged (lines 5-6) and the variables that indicate the amount of assets that will have to be deposited (lines 7-8).

From line 9 to line 17, the global variables are valued using the values contained in the function parameters. In particular:

1. Lines 10-13: information about the parties to the contracts is stored (public key and address);
2. Lines 14-17: the variables indicating the quantity of assets that must be deposited in the contract by each participant in the contract are set.

From line 18 to line 19 the body of the function is defined, which in this case is empty, and in line 20 the end of the function is indicated by the function `HALT`.

Deposit of the first asset

This portion of code allows Alice to deposit a certain amount of assets, agreed during the *agreement* phase. In particular:

1. Line 10: this function can only be called by Alice and if the contract is in the `@Inactive` state. Note that this function takes a `asset` as an argument (note `[y]`);

2. Line 11: a check is made to verify if the quantity received as input is equal to the quantity established in the *agreement* phase;
3. Line 12: this instruction represents the deposit of a certain quantity of assets within the instance of the contract;
4. Line 14: at the end of the function, the state of the contract will change from `@Inactive` to `@Swap`.

```
10    @Inactive Alice : depositAssetA()[y]
11          (y == amountAssetA) {
12              y -o assetA;
13              _
14      } ==> @Swap
```

The following bytecode is associated with this function in *Stipula*:

```
21    fn Inactive Alice depositAssetA Swap asset
22    args:
23    PUSH asset :y
24    AINST asset :y
25    ASTORE y
26    start:
27    ALOAD y
28    GLOAD amountAssetA
29    ISEQ
30    JMPIF if_branch
31    RAISE AMOUNT_NOT_EQUAL
32    JMP end
33    if_branch:
34    ALOAD y
35    GLOAD assetA
36    DEPOSIT assetA
37    end:
38    HALT
```

On line 21 it is possible to note the signature of the function, where the following are specified:

1. The state the contract instance must be in in order to call this function (`@Inactive`);
2. The party that can call this function (`Alice`);
3. The name of the function (`depositAssetA`);
4. The state the contract instance will go to once the function's execution has finished without errors (`Swap`);
5. The type of the function parameter (`asset`).

From line 22 to line 25, the function argument is instantiated. This variable is stored in the argument space (`argumentSpace`).

From line 26 to line 37 is the body of the function. In particular, from line 27 to line 30, the virtual machine checks if the quantity of assets received as input is equal

to that established during the *agreement* phase. If the result of this check is *false*, then the virtual machine will continue executing first with line 31 and then with line 32, the function execution will terminate. If instead the result of the check is *true*, starting from line 30, the virtual machine will execute the instructions starting from line 33 in sequence.

From line 34 to line 36, it is possible to note the effective action of *deposit* of assets within the instance of the contract (*Pay-to-Contract*). In particular:

```
33    ...
34    ALOAD y
35    GLOAD assetA
36    DEPOSIT assetA
37    ...
```

corresponds to the following line written in *Stipula*

```
11    ...
12              y -o assetA;
13    ...
```

Deposit of the second asset and swap

The code of this function is very similar to that of the previous function, except for the *swap* operation. This function, in fact, allows Bob to deposit the asset in his possession and then to exchange the assets between the participants of the contract. In particular, it is possible to observe that line 19 and line 20 implement the actual asset swap operation, ie: the asset previously deposited by Alice is sent to Bob; the asset deposited in this function by Bob is sent to Alice.

```
16    @Swap Bob : depositAssetBAndSwap()[y]
17          (y == amountAssetB) {
18              y -o assetB
19              assetB -o Alice
20              assetA -o Bob;
21              _
22      } ==> @End
23    }
```

The following bytecode is associated with this function in *Stipula*:

```
39    fn Swap Bob depositAssetBAndSwap End asset
40    args:
41    PUSH asset :y
42    AINST asset :y
43    ASTORE y
44    start:
45    ALOAD y
46    GLOAD amountAssetB
47    ISEQ
48    JMPIF if_branch
49    RAISE AMOUNT_NOT_EQUAL
50    JMP end
```

```
51    if_branch:
52    ALOAD y
53    GLOAD assetB
54    DEPOSIT assetB
55    PUSH real 100 2
56    GLOAD assetB
57    GLOAD Alice
58    WITHDRAW assetB
59    PUSH real 100 2
60    GLOAD assetA
61    GLOAD Bob
62    WITHDRAW assetA
63    end:
64    HALT
```

Again, the bytecode produced is very similar to that produced for the previous function. It can be seen that from line 55 to line 62 the asset swap is implemented. In particular, it is possible to note:

1. From line 52 to line 54 there is a *deposit* (*Pay-to-Contract*)

   ```
   51      ...
   52      ALOAD y
   53      GLOAD assetB
   54      DEPOSIT assetB
   55      ...
   ```

   This piece of code corresponds to the following line written in *Stipula*

   ```
   17      ...
   18              y -o assetB;
   19      ...
   ```

2. From line 55 to line 58 there is a *withdraw* towards Alice (*Pay-to-Party*)

   ```
   54      ...
   55      PUSH real 100 2
   56      GLOAD assetB
   57      GLOAD Alice
   58      WITHDRAW assetB
   59      ...
   ```

   This piece of code corresponds to the following line written in *Stipula*

   ```
   18      ...
   19              assetB -o Alice;
   20      ...
   ```

3. From line 59 to line 62 there is a *withdraw* towards Bob (*Pay-to-Party*)

   ```
   58      ...
   59      PUSH real 100 2
   60      GLOAD assetA
   61      GLOAD Bob
   ```

```
62        WITHDRAW assetA
63        ...
```

This piece of code corresponds to the following line written in *Stipula*

```
19        ...
20                assetA -o Bob;
21        ...
```

Example of execution

An example of execution of this contract is illustrated.

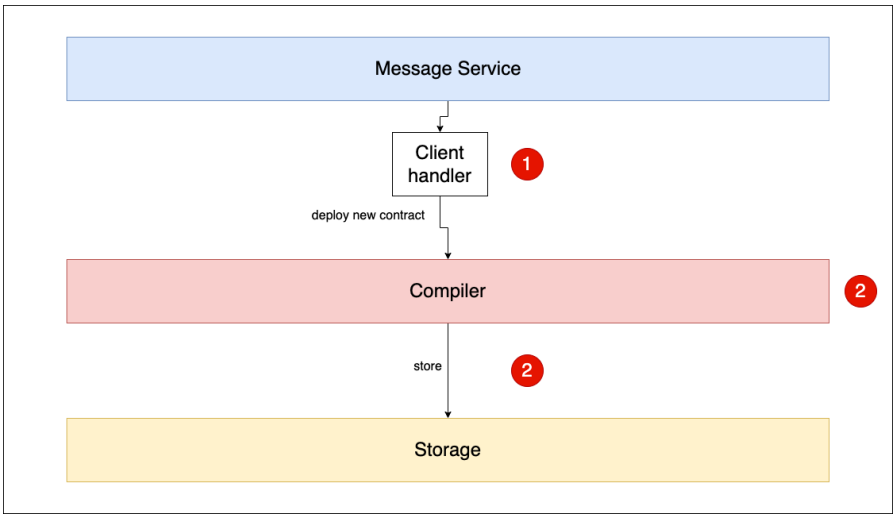


Figure 4.8: New contract upload execution flow.

Deploy contract The flow for *deploy* a new contract is illustrated in figure 4.8. The request is received by the `MessageService` and via the `ClientHandler` the request is directed to the compiler (**1**). The contract is compiled and if the compilation returns no errors (**2**), it stores the contract and the compiled in the *Storage* module (**3**).

Request to the server for contract deployment:

```
1       {
2         "message": {
3           "sourceCode": "stipula SwapAsset {\n    asset
            ↪  assetA:stipula_assetA_ed8i9wk,
            ↪  assetB:stipula_assetB_pl1n5cc\n    field amountAssetA,
            ↪  amountAssetB\n    init Inactive\n\n    agreement (Alice,
            ↪  Bob)(amountAssetA, amountAssetB) {\n        Alice, Bob:
            ↪  amountAssetA, amountAssetB\n    } ==> @Inactive\n\n
            ↪  @Inactive Alice : depositAssetA()[y]\n        (y ==
            ↪  amountAssetA) {\n            y -o assetA;\n            _\n
            ↪    } ==> @Swap\n\n    @Swap Bob :
            ↪  depositAssetBAndSwap()[y]\n        (y == amountAssetB) {\n
            ↪            y -o assetB\n            assetB -o Alice\n
            ↪      assetA -o Bob;\n            _\n    } ==> @End\n}",
```

```
        "type": "DeployContract"
      },
      "signatures": {
        "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCo/GjVKS+3gAA55+kko41yIN
        ↪ dOcCLQMSBQyuTTkKHE1mhu/TgOpivM0wLPsSga8hQMr3+v3aR0IF/vfCRf6
        ↪ SdiXmWx/jflmEXtnT6fkGcnV6dGNUpHWXSpwUIDt0N88jfnEqekx4S+KDCK
        ↪ g99sGEeHeT65fKS8lB0gjHMt9AOriwIDAQAB":
        ↪ "V5gJHSax5J5nWYZlyhJr+RdJhbWrog9/urvyfWPTNWf6jkLRT16xAdLYBR
        ↪ 1NucOmKTf9iW6mVMVpUxtrGPXktTUEIzxJpp81jR06hDBUpH0Eu6pkiw9no
        ↪ mTUZvuCX9DR/+WOSBz0jMO5l0znl6At3OP1mXsgNyRtPJTi2q4yHs0="
      }
    }
```

Server response:

```
    {
      "data": "d50ed1a3-7a65-4238-867e-df48536b7243",
      "statusCode": 200,
      "statusMessage": "Success",
      "type": "SuccessDataResponse"
    }
```

The value in the `data` field indicates the identifier of the deployed contract.

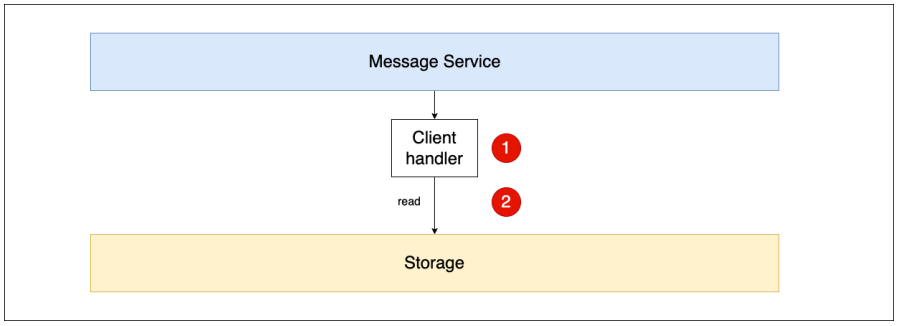


Figure 4.9: Execution flow for Alice's funds read request.

Single-use-seals by Alice The description of the following flow is illustrated in figure 4.9. When a user wants to know the available funds associated with his address, the user sends a particular message (**1**). The request is directed to the *Storage* module (**2**) and the availability of funds is sent to the user in response.

Request to the server to get the single-use-seals in Alice's possession:

```
    {
      "message": {
        "address": "ubL35Am7TimL5R4oMwm2OxgAYA3XT3BeeDE56oxqdLc=",
        "type": "GetOwnershipsByAddress"
      },
      "signatures": {
```

```
          "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCo/GjVKS+3gAA55+kko41yIN
          ↪ dOcCLQMSBQyuTTkKHE1mhu/TgOpivM0wLPsSga8hQMr3+v3aR0IF/vfCRf6
          ↪ SdiXmWx/jflmEXtnT6fkGcnV6dGNUpHWXSpwUIDt0N88jfnEqekx4S+KDCK
          ↪ g99sGEeHeT65fKS8lB0gjHMt9AOriwIDAQAB":
          ↪ "MomZTc63z7PfH35c1dL4tjXebcsW+0Zxl0nP1NQdcUFws98DX+bMWI7L0C
          ↪ 6IO5lxvkYve4zdio1Crn97FXvngK4aVfiEZEnHOJ0tstq7uQYGErM3DDAAB
          ↪ qPq8HH5yoKnLST2LWpO0oD8G/VXvIE6qMT5D34W1Ci0q4uh+7y3EcY="
        }
      }
```

Server response:

```
      {
        "data": "[
            Ownership{
              id='2b4a4614-3bb4-4554-93fe-c034c3ba5a9c',
              singleUseSeal=SingleUseSeal{
                assetId='stipula_assetA_ed8i9wk',
                amount=RealType{
                  value=1400,
                  decimals=2
                },
                lockScript='DUP\nSHA256\nPUSH str ubL35Am7TimL5R4oMwm2Oxg
                ↪  AYA3XT3BeeDE56oxqdLc=\nEQUAL\nCHECKSIG\nHALT\n'
              },
              unlockScript='',
              contractInstanceId=''
            }
          ]",
        "statusCode": 200,
        "statusMessage": "Success",
        "type": "SuccessDataResponse"
      }
```

Single-use-seals by Bob Server request to get Bob's single-use-seals:

```
      {
        "message": {
          "address": "f3hVW1Amltnqe3KvOT00eT7AU23FAUKdgmCluZB+nss=",
          "type": "GetOwnershipsByAddress"
        },
        "signatures": {
          "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDErzzgD2ZslZxciFAiX3/ot7
          ↪ lrkZDw4148jFZrsDZPE6CVs9xXFSHGgy/mFvIFLXhnChO6Nyd2be3lbgeav
          ↪ LMCMVUiTStXr117Km17keWpb3sItkKKsLFBOcIIU8XXowI/OhzQN2XPZYES
          ↪ HgjdQ5vwEj2YyueiS7WKP94YWz/pswIDAQAB":
          ↪ "hSNodnUyusffNlv+KNq4605pFvqh91pVspFhTgbmWccE/LKM6h4bedpvTg
          ↪ MHoVDezvA7v2XTzmLG5eL3lOeA6I2xJMH32DcV60IPSoh61oVHnwPQcQHYO
          ↪ 39D4y5VSJ0GMQJKIcTEq3fqIdabg7261xUaegHUnXrcyynh9GpMJxk="
        }
      }
```

Server response:

```
      {
        "data": "[
            Ownership{
              id='7a19f50e-eae9-461d-bd58-9946ea39ccf0',
```

```
5               singleUseSeal=SingleUseSeal{
6                 assetId='stipula_assetB_pl1n5cc',
7                 amount=RealType{
8                   value=1100,
9                   decimals=2
10                },
11                lockScript='DUP\nSHA256\nPUSH str f3hVW1Amltnqe3KvOT00eT7
                  ↪  AU23FAUKdgmCluZB+nss=\nEQUAL\nCHECKSIG\nHALT\n'
12              },
13              unlockScript='',
14              contractInstanceId=''
15            }
16          ]",
17        "statusCode": 200,
18        "statusMessage": "Success",
19        "type": "SuccessDataResponse"
20      }
```

Agreement The description of the following flow is illustrated in figure 4.5. When users have agreed to execute an agreement, an `AgreementCall` message is sent (see 4.3.4). This request is placed in the request queue (**1.1** and **1.2**). The virtual machine dequeues the request (**2**) and the function *agreement* (**4**) is executed. Once the execution of the function has finished without errors, the result of the processing will be stored in the *Storage* module (**5**) and finally the virtual machine will notify the client of the success of the operation (**6**).

Request to the server to make the `agreement` function call:

```
1       {
2         "message": {
3           "contractId": "d50ed1a3-7a65-4238-867e-df48536b7243",
4           "arguments": [
5             {
6               "argument": {
7                 "first": "real",
8                 "second": "amountAssetA",
9                 "third": "1400 2"
10              }
11            },
12            {
13              "argument": {
14                "first": "real",
15                "second": "amountAssetB",
16                "third": "1100 2"
17              }
18            }
19          ],
20          "parties": {
21            "Bob": {
22              "address": "f3hVW1Amltnqe3KvOT00eT7AU23FAUKdgmCluZB+nss=",
23              "publicKey":
                ↪  "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDErzzgD2ZslZxciF
                ↪  AiX3/ot7lrkZDw4148jFZrsDZPE6CVs9xXFSHGgy/mFvIFLXhnChO6N
                ↪  yd2be3lbgeavLMCMVUiTStXr117Km17keWpb3sItkKKsLFBOcIIU8XX
                ↪  owI/OhzQN2XPZYESHgjdQ5vwEj2YyueiS7WKP94YWz/pswIDAQAB"
```

```
24            },
25            "Alice": {
26              "address": "ubL35Am7TimL5R4oMwm2OxgAYA3XT3BeeDE56oxqdLc=",
27              "publicKey":
                ↪  "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCo/GjVKS+3gAA55+
                ↪  kko41yINdOcCLQMSBQyuTTkKHE1mhu/TgOpivM0wLPsSga8hQMr3+v3
                ↪  aR0IF/vfCRf6SdiXmWx/jflmEXtnT6fkGcnV6dGNUpHWXSpwUIDt0N8
                ↪  8jfnEqekx4S+KDCKg99sGEeHeT65fKS8lB0gjHMt9AOriwIDAQAB"
28            }
29          },
30          "type": "AgreementCall"
31        },
32        "signatures": {
33          "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDErzzgD2ZslZxciFAiX3/ot7
            ↪  lrkZDw4148jFZrsDZPE6CVs9xXFSHGgy/mFvIFLXhnChO6Nyd2be3lbgeav
            ↪  LMCMVUiTStXr117Km17keWpb3sItkKKsLFBOcIIU8XXowI/OhzQN2XPZYES
            ↪  HgjdQ5vwEj2YyueiS7WKP94YWz/pswIDAQAB":
            ↪  "crMKGFVc5QYmYfbyxDaqhXEi0/GRO+j2OD8HtBbysVm1/+2D+nFATAOvm+
            ↪  LbDtLMMBHxTE8a4JHzMN1DZ1uokkwHKyv80/IVMLwjZi6RFl1Jk7jUpUq6n
            ↪  BCPfqfa7u2IKtzv0joJXR/8BNyN3u6+PReS+4N530+ESN3W2P3tIFk=",
34          "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCo/GjVKS+3gAA55+kko41yIN
            ↪  dOcCLQMSBQyuTTkKHE1mhu/TgOpivM0wLPsSga8hQMr3+v3aR0IF/vfCRf6
            ↪  SdiXmWx/jflmEXtnT6fkGcnV6dGNUpHWXSpwUIDt0N88jfnEqekx4S+KDCK
            ↪  g99sGEeHeT65fKS8lB0gjHMt9AOriwIDAQAB":
            ↪  "OK9fuuEHTIV5gjtghgvFqsJZI98Ip7IYXvph0J79kTwfRVvJnH5mX9Rs/l
            ↪  DUWnznOmY3HTADwn4QgzMQgdu+qAfixoyJWvZJZ8XjNo/N1YI3nnaaXhvkp
            ↪  R8OSHhxqhFLfET6rAx5qXpziOZS7NfcIasn6Lj35hbQCfcjKvxf76w="
35        }
36      }
```

Server response:

```
1       {
2         "data": "e9cbb96e-4d20-47d2-80e6-5b56701800b1",
3         "statusCode": 200,
4         "statusMessage": "Success",
5         "type": "SuccessDataResponse"
6       }
```

The value in the `data` field indicates the identifier of the created contract instance.

depositAssetA call In this function Alice has to make a *Pay-to-Contract*. Compared to the description of the previous flow, before the *Legal Contract Virtual Machine* executes the function, it is necessary to check that the *single-use-seal* sent by Alice actually belongs to Alice and is of the quantity requested by the instance of the contract. To do this, it is necessary to carry out these checks with the *Script Virtual Machine* (see point **3** of figure 4.5). Once these checks have been completed, the virtual machine will be able to proceed with the execution of the function requested by Alice.

Request to the server to make the `depositAssetA` function call:

```
1       {
2         "message": {
3           "contractInstanceId": "e9cbb96e-4d20-47d2-80e6-5b56701800b1",
4           "functionName": "depositAssetA",
5           "arguments": [
6             {
```

```
7               "argument": {
8                 "first": "asset",
9                 "second": "y",
10                "third": {
11                  "ownershipId": "2b4a4614-3bb4-4554-93fe-c034c3ba5a9c",
12                  "address":
                    ↪  "ubL35Am7TimL5R4oMwm2OxgAYA3XT3BeeDE56oxqdLc=",
13                  "unlockScript": "PUSH str PLjodnT+m3RNIitQAPBDCsRmJPHCq
                    ↪  rwZOY/CPiHFZGnl+DRN6soqxMy3ehTFaUwxBjjf7qfBfvTDq5oB
                    ↪  ItTFrtz1Rn5SDS1ybdbkwpKaOXVglNOw7ZEG9bbZ1mo1oA7IAjR
                    ↪  iIilzUetCstE5rPZIf9XOXr/RQ5AHkZUn2CztsvA=\nPUSH str
                    ↪  MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCo/GjVKS+3gAA
                    ↪  55+kko41yINdOcCLQMSBQyuTTkKHE1mhu/TgOpivM0wLPsSga8h
                    ↪  QMr3+v3aR0IF/vfCRf6SdiXmWx/jflmEXtnT6fkGcnV6dGNUpHW
                    ↪  XSpwUIDt0N88jfnEqekx4S+KDCKg99sGEeHeT65fKS8lB0gjHMt
                    ↪  9AOriwIDAQAB\n"
14                }
15              }
16            }
17          ],
18          "type": "FunctionCall"
19        },
20        "signatures": {
21          "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCo/GjVKS+3gAA55+kko41yIN
            ↪  dOcCLQMSBQyuTTkKHE1mhu/TgOpivM0wLPsSga8hQMr3+v3aR0IF/vfCRf6
            ↪  SdiXmWx/jflmEXtnT6fkGcnV6dGNUpHWXSpwUIDt0N88jfnEqekx4S+KDCK
            ↪  g99sGEeHeT65fKS8lB0gjHMt9AOriwIDAQAB":
            ↪  "MVm0fv9zBntC7ElPhNYaISpgmOdCh8blRsvkU2gtulbWQvwg/CuKtcOIHx
            ↪  akTrffnrW7iw/KLB0n46HulBL6KAcl02U9HSt0+YwX3imJ50QVWU7kmLoMy
            ↪  5d8uQ+seZzXifsaf7OvE1OpAWXNwh7ICsRZv9U6aV39c13SUqwHjTs="
22        }
23      }
```

Server response:

```
1       {
2         "statusCode": 200,
3         "statusMessage": "Success",
4         "type": "SuccessDataResponse"
5       }
```

depositAssetBAndSwap call Request to the server to make the `depositAssetBAndSwap` function call:

```
1       {
2         "message": {
3           "contractInstanceId": "e9cbb96e-4d20-47d2-80e6-5b56701800b1",
4           "functionName": "depositAssetBAndSwap",
5           "arguments": [
6             {
7               "argument": {
8                 "first": "asset",
9                 "second": "y",
10                "third": {
11                  "ownershipId": "7a19f50e-eae9-461d-bd58-9946ea39ccf0",
12                  "address":
                    ↪  "f3hVW1Amltnqe3KvOT00eT7AU23FAUKdgmCluZB+nss=",
```

```
                "unlockScript": "PUSH str Q0bPh9lThyrg1slz9AGDJDJh1BecN
                ↪  9SlGCeVe3BqLod+zO7q0wvIy8tLognHNBkR8e8zKo6nWGQ8qZ7e
                ↪  gjOmm5BQsqZzt8xL3gBbR36vgk9J3G9ObiTR2Dd7hMqsqyJnLT3
                ↪  aZUPXGc6RZoM/iUFGJUXhq2T6DStvYNKuAH+Lfow=\nPUSH str
                ↪  MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDErzzgD2ZslZx
                ↪  ciFAiX3/ot7lrkZDw4148jFZrsDZPE6CVs9xXFSHGgy/mFvIFLX
                ↪  hnChO6Nyd2be3lbgeavLMCMVUiTStXr117Km17keWpb3sItkKKs
                ↪  LFBOcIIU8XXowI/OhzQN2XPZYESHgjdQ5vwEj2YyueiS7WKP94Y
                ↪  Wz/pswIDAQAB\n"
              }
            }
          }
        ],
        "type": "FunctionCall"
      },
      "signatures": {
        "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDErzzgD2ZslZxciFAiX3/ot7
        ↪  lrkZDw4148jFZrsDZPE6CVs9xXFSHGgy/mFvIFLXhnChO6Nyd2be3lbgeav
        ↪  LMCMVUiTStXr117Km17keWpb3sItkKKsLFBOcIIU8XXowI/OhzQN2XPZYES
        ↪  HgjdQ5vwEj2YyueiS7WKP94YWz/pswIDAQAB":
        ↪  "kh7JupouiEdeLuilXUdoJqAuPVx28JTg9dySp/ZNJGD5+XW8YhhIgiMJYO
        ↪  hGeN6DJTj/x+TmC96uyS8IwssUt/Hulnh2OAZzkc3FljWj1k/XfLOyye95u
        ↪  +YBxg+t8AddQBi+4uA4yOdzb8YdrONlzGu7t0roirmO8SbOqQR1uX8="
      }
    }
```

Server response:

```
    {
      "statusCode": 200,
      "statusMessage": "Success",
      "type": "SuccessDataResponse"
    }
```

Single-use-seals by Alice Server request to get Alice's single-use-seals:

```
    {
      "message": {
        "address": "ubL35Am7TimL5R4oMwm2OxgAYA3XT3BeeDE56oxqdLc=",
        "type": "GetOwnershipsByAddress"
      },
      "signatures": {
        "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCo/GjVKS+3gAA55+kko41yIN
        ↪  dOcCLQMSBQyuTTkKHE1mhu/TgOpivM0wLPsSga8hQMr3+v3aR0IF/vfCRf6
        ↪  SdiXmWx/jflmEXtnT6fkGcnV6dGNUpHWXSpwUIDt0N88jfnEqekx4S+KDCK
        ↪  g99sGEeHeT65fKS8lB0gjHMt9AOriwIDAQAB":
        ↪  "MomZTc63z7PfH35c1dL4tjXebcsW+0Zxl0nP1NQdcUFws98DX+bMWI7L0C
        ↪  6IO5lxvkYve4zdio1Crn97FXvngK4aVfiEZEnHOJ0tstq7uQYGErM3DDAAB
        ↪  qPq8HH5yoKnLST2LWpO0oD8G/VXvIE6qMT5D34W1Ci0q4uh+7y3EcY="
      }
    }
```

Server response:

```
    {
      "data": "[
          Ownership{
            id='2b4a4614-3bb4-4554-93fe-c034c3ba5a9c',
```

```
5               singleUseSeal=SingleUseSeal{
6                 assetId='stipula_assetA_ed8i9wk',
7                 amount=RealType{
8                   value=1400,
9                   decimals=2
10                },
11                lockScript='DUP\nSHA256\nPUSH str ubL35Am7TimL5R4oMwm2Oxg
                    ↪  AYA3XT3BeeDE56oxqdLc=\nEQUAL\nCHECKSIG\nHALT\n'
12              },
13              unlockScript='PUSH str PLjodnT+m3RNIitQAPBDCsRmJPHCqrwZOY/C
                  ↪  PiHFZGnl+DRN6soqxMy3ehTFaUwxBjjf7qfBfvTDq5oBItTFrtz1Rn5
                  ↪  SDS1ybdbkwpKaOXVglNOw7ZEG9bbZ1mo1oA7IAjRiIilzUetCstE5rP
                  ↪  ZIf9XOXr/RQ5AHkZUn2CztsvA=\nPUSH str
                  ↪  MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCo/GjVKS+3gAA55+k
                  ↪  ko41yINdOcCLQMSBQyuTTkKHE1mhu/TgOpivMOwLPsSga8hQMr3+v3a
                  ↪  ROIF/vfCRf6SdiXmWx/jflmEXtnT6fkGcnV6dGNUpHWXSpwUIDt0N88
                  ↪  jfnEqekx4S+KDCKg99sGEeHeT65fKS8lBOgjHMt9AOriwIDAQAB\n',
14              contractInstanceId='e9cbb96e-4d20-47d2-80e6-5b56701800b1'
15            },
16            Ownership{
17              id='4cbec85d-f17e-4928-a029-7cf0e646a3f6',
18              singleUseSeal=SingleUseSeal{
19                assetId='stipula_assetB_pl1n5cc',
20                amount=RealType{
21                  value=100,
22                  decimals=2
23                },
24                lockScript='DUP\nSHA256\nPUSH str ubL35Am7TimL5R4oMwm2Oxg
                    ↪  AYA3XT3BeeDE56oxqdLc=\nEQUAL\nCHECKSIG\nHALT\n'
25              },
26              unlockScript='',
27              contractInstanceId=''
28            }
29          ]",
30        "statusCode": 200,
31        "statusMessage": "Success",
32        "type": "SuccessDataResponse"
33      }
```

It is possible to see that the first single-use-seal has been spent and that's what was deposited in the contract instance. Evidence that the funds have been spent is given by the `unlockScript` field. While, the second single-use-seal represents the asset that was in Bob's possession.

Single-use-seals by Bob Server request to get Bob's single-use-seals:

```
1       {
2         "message": {
3           "address": "f3hVW1Amltnqe3KvOT00eT7AU23FAUKdgmCluZB+nss=",
4           "type": "GetOwnershipsByAddress"
5         },
6         "signatures": {
```

```
          "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDErzzgD2ZslZxciFAiX3/ot7lrkZDw4148jFZrsDZPE6CVs9xXFSHGgy/mFvIFLXhnChO6Nyd2be3lbgeavLMCMVUiTStXr117Km17keWpb3sItkKKsLFBOcIIU8XXowI/OhzQN2XPZYESHgjdQ5vwEj2YyueiS7WKP94YWz/pswIDAQAB":
          "hSNodnUyusffNlv+KNq4605pFvqh91pVspFhTgbmWccE/LKM6h4bedpvTgMHoVDezvA7v2XTzmLG5eL3lOeA6I2xJMH32DcV60IPSoh61oVHnwPQcQHY039D4y5VSJOGMQJKIcTEq3fqIdabg7261xUaegHUnXrcyynh9GpMJxk="
        }
      }
```

Server response:

```
      {
        "data": "[
            Ownership{
              id='7a19f50e-eae9-461d-bd58-9946ea39ccf0',
              singleUseSeal=SingleUseSeal{
                assetId='stipula_assetB_pl1n5cc',
                amount=RealType{
                  value=1100,
                  decimals=2
                },
                lockScript='DUP\nSHA256\nPUSH str f3hVW1Amltnqe3KvOT0OeT7AU23FAUKdgmCluZB+nss=\nEQUAL\nCHECKSIG\nHALT\n'
              },
              unlockScript='PUSH str Q0bPh9lThyrg1slz9AGDJDJh1BecN9SlGCeVe3BqLod+zO7q0wvIy8tLognHNBkR8e8zKo6nWGQ8qZ7egjOmm5BQsqZzt8xL3gBbR36vgk9J3G9ObiTR2Dd7hMqsqyJnLT3aZUPXGc6RZoM/iUFGJUXhq2T6DStvYNKuAH+Lfow=\nPUSH str MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDErzzgD2ZslZxciFAiX3/ot7lrkZDw4148jFZrsDZPE6CVs9xXFSHGgy/mFvIFLXhnChO6Nyd2be3lbgeavLMCMVUiTStXr117Km17keWpb3sItkKKsLFBOcIIU8XXowI/OhzQN2XPZYESHgjdQ5vwEj2YyueiS7WKP94YWz/pswIDAQAB\n',
              contractInstanceId='e9cbb96e-4d20-47d2-80e6-5b56701800b1'
            },
            Ownership{
              id='bd1f5959-cd8d-4716-8ece-19e1757c6ac2',
              singleUseSeal=SingleUseSeal{
                assetId='stipula_assetA_ed8i9wk',
                amount=RealType{
                  value=100,
                  decimals=2
                },
                lockScript='DUP\nSHA256\nPUSH str f3hVW1Amltnqe3KvOT0OeT7AU23FAUKdgmCluZB+nss=\nEQUAL\nCHECKSIG\nHALT\n'
              },
              unlockScript='',
              contractInstanceId=''
            }
          ]",
        "statusCode": 200,
        "statusMessage": "Success",
        "type": "SuccessDataResponse"
      }
```

It is possible to see that the first single-use-seal has been spent and that's what was deposited in the contract instance. Evidence that the funds have been spent is given

by the `unlockScript` field. While, the second single-use-seal represents the asset that was in Alice's possession.

4.8.2 Asset swap with scheduled event

The complete code is present in the appendix A.2.1. The *Stipula* code compared to the previous version does not change much. The only changes made are:

1. Line 3: A new global variable `waitTimeBeforeSwapping` is defined. This variable indicates the time needed to wait before being able to swap assets. It is a value that is agreed between the participants of the *agreement* contract;

2. Lines 6 and 7: it is specified that a value for `waitTimeBeforeSwapping` must be supplied during the *agreement* phase;

3. Line 14: the state the contract instance will go to once the function is executed `depositAssetA` will exit without errors, it is no longer `@Swap` but `@Deposit`;

4. Line 16: if Bob wants to deposit his asset, the contract instance must be in the `@Deposit` and no longer `@Swap`. Also, the function name changes from `depositAssetBAndSwap` to `depositAssetB`;

5. From line 19 to line 23: the code that encodes the *obligation* that will have to be executed at the time indicated in line 19 is defined, that is, an event will be scheduled that will execute the obligation at the time `now + waitTimeBeforeSwapping`;

6. Line 24: the state the contract instance will go to once the `depositAssetB` will exit without errors, it is no longer `@End` but `@Swap`;

7. Line 20: in order to execute the obligation at the defined time, the status of the contract instance must be `@Swap`;

8. Line 23: the state in which the contract instance will go once the execution of the obligation has finished without errors, will be `@End`.

The bytecode is almost similar to the one produced for the previous example, the substantial change occurs for the encoding of the *obligation*. In particular:

1. Lines 58 to 60: This piece of code is part of the `depositAssetB` function. These specific lines allow to calculate the *absolute* time, necessary to schedule an *event*, which will carry out a particular function call. To indicate where the function code to be executed by the event begins, the `TRIGGER obligation_1` instruction is used. Once the execution of the function is finished, the event will be scheduled and when the time t arrives, the `EventTrigger` will put the request in the request queue (see 4.6.6);

2. Line 64 to line 75: lines 66 to 75 correspond exactly to lines 55 to 64 of the previous function. However, this code is now part of a particular function, whose signature is defined on line 64. Indeed, it specifies: this code encodes a *obligation* (`obligation`); in order to perform this obligation, the state of the contract instance must be `@Swap`; the name of the function `obligation_1`; the state in which the contract instance will go once the execution of the obligation has finished without errors, will be `@End`.

Example of execution

An example of execution of this contract is illustrated in the appendix A.2.2. Only a few steps will be shown in this section.

Agreement The *agreement* phase is very similar to the previous example, the only change is to set the value to the variable `waitTimeBeforeSwapping`.

Request to the server to make the `agreement` function call:


```
{
  "message": {
    "contractId": "79caadf1-abbe-418a-a9a2-bd132a6f3e9e",
    "arguments": [
      {
        "argument": {
          "first": "real",
          "second": "amountAssetA",
          "third": "1400 2"
        }
      },
      {
        "argument": {
          "first": "real",
          "second": "amountAssetB",
          "third": "1100 2"
        }
      },
      {
        "argument": {
          "first": "time",
          "second": "waitTimeBeforeSwapping",
          "third": "100"
        }
      }
    ],
    "parties": {
      "Bob": {
        "address": "f3hVW1Amltnqe3KvOT00eT7AU23FAUKdgmCluZB+nss=",
        "publicKey":
        ↪  "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDErzzgD2ZslZxciF
        ↪  AiX3/ot7lrkZDw4148jFZrsDZPE6CVs9xXFSHGgy/mFvIFLXhnChO6N
        ↪  yd2be3lbgeavLMCMVUiTStXr117Km17keWpb3sItkKKsLFBOcIIU8XX
        ↪  owI/OhzQN2XPZYESHgjdQ5vwEj2YyueiS7WKP94YWz/pswIDAQAB"
      },
      "Alice": {
        "address": "ubL35Am7TimL5R4oMwm2OxgAYA3XT3BeeDE56oxqdLc=",
        "publicKey":
        ↪  "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCo/GjVKS+3gAA55+
        ↪  kko41yINdOcCLQMSBQyuTTkKHE1mhu/TgOpivMOwLPsSga8hQMr3+v3
        ↪  aROIF/vfCRf6SdiXmWx/jflmEXtnT6fkGcnV6dGNUpHWXSpwUIDt0N8
        ↪  8jfnEqekx4S+KDCKg99sGEeHeT65fKS8lBOgjHMt9AOriwIDAQAB"
      }
    },
    "type": "AgreementCall"
  },
```

```
39        "signatures": {
40          "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDErzzgD2ZslZxciFAiX3/ot7
            ↪ lrkZDw4148jFZrsDZPE6CVs9xXFSHGgy/mFvIFLXhnChO6Nyd2be3lbgeav
            ↪ LMCMVUiTStXr117Km17keWpb3sItkKKsLFBOcIIU8XXowI/OhzQN2XPZYES
            ↪ HgjdQ5vwEj2YyueiS7WKP94YWz/pswIDAQAB":
            ↪ "Wrqyz5udZAGarLbSlxhYD+Ur6+EqTCFiwqBHEL2IsO5Y23Yxv14O3Uzknr
            ↪ wK41L5LPUgVxR3K75AAZ4n+UcUdDNHlm9KHN7rqpsbe7v3yK2q8Qkk6c4IY
            ↪ NPDRFy3Zw62HH94O7tx8CzcvRfdX4fi+RItf4Fa7hb8Ui/crxDEQN8=",
41          "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCo/GjVKS+3gAA55+kko41yIN
            ↪ dOcCLQMSBQyuTTkKHE1mhu/TgOpivMOwLPsSga8hQMr3+v3aROIF/vfCRf6
            ↪ SdiXmWx/jflmEXtnT6fkGcnV6dGNUpHWXSpwUIDt0N88jfnEqekx4S+KDCK
            ↪ g99sGEeHeT65fKS8lBOgjHMt9AOriwIDAQAB":
            ↪ "o/bdsudfHdR4BBd9EVaGYikksIezSEdwhHELH/f7xRD9g4uokO5g8wHph6
            ↪ LOht5dt9Y+dYt+Qrt+zNZzGUP8a5OR7WB2gNzOJn3zndKnVoBVhsda/zEwI
            ↪ A2pqccP2Sda7zCYiFTfgnmlUZZZfxjtLazBUzDE/vVVFcwtXAHYMXk="
42        }
43      }
```

For simplicity, the value for `waitTimeBeforeSwapping` is equal to 100, that is, after Bob deposits his asset, they will wait 100 seconds before exchanging assets.

Server response:

```
1       {
2         "data": "48819afd-e28f-4037-82fd-1d073ee1d318",
3         "statusCode": 200,
4         "statusMessage": "Success",
5         "type": "SuccessDataResponse"
6       }
```

The value in the `data` field indicates the identifier of the created contract instance.

Event trigger and execution of the obligation The call of `depositAssetA` and `depositAssetB` are the same as the previous example.

From the server logs it can be seen that the event was triggered, the code encoding the obligation was loaded and executed:

```
1       EventTrigger: A new scheduled request has been triggered =>
        ↪  EventTriggerSchedulingRequest{
2         request=CreateEventRequest{
3           obligationFunctionName='obligation_1',
4           time=1680032647
5         },
6         contractId='79caadf1-abbe-418a-a9a2-bd132a6f3e9e',
7         contractInstanceId='48819afd-e28f-4037-82fd-1d073ee1d318'
8       }
9       EventTrigger: Enqueuing the request...
10      EventTrigger: Notifying the virtual machine...
11      EventTrigger: Virtual machine notified
12      EventTrigger: Removing the request from EventTriggerHandler...
13      VirtualMachine: Ready to dequeue a value...
14      VirtualMachine: Request received => Pair{
15        first=null,
16        second=EventTriggerSchedulingRequest{
17          request=CreateEventRequest{
18            obligationFunctionName='obligation_1',
19            time=1680032647
20          },
```

```
21          contractId='79caadf1-abbe-418a-a9a2-bd132a6f3e9e',
22          contractInstanceId='48819afd-e28f-4037-82fd-1d073ee1d318'
23        }
24      }
25      VirtualMachine: Just received a trigger request
26      loadObligationFunction: Loading the obligation function...
27      loadObligationFunction: Obligation function loaded
28      VirtualMachine: Function
29      start:
30      PUSH real 100 2
31      GLOAD assetB
32      GLOAD Alice
33      WITHDRAW assetB
34      PUSH real 100 2
35      GLOAD assetA
36      GLOAD Bob
37      WITHDRAW assetA
38      end:
39      HALT
40
41      loadBytecode: Loading the bytecode...
42      loadBytecode: Bytecode loaded
43
44      VirtualMachine: loadBytecode
45      start:
46      PUSH real 100 2
47      GLOAD assetB
48      GLOAD Alice
49      WITHDRAW assetB
50      PUSH real 100 2
51      GLOAD assetA
52      GLOAD Bob
53      WITHDRAW assetA
54      end:
55      HALT
56
57      LegalContractVirtualMachine: execute => Final state of the
        ↪  execution below
58      LegalContractVirtualMachine: execute => The stack is empty
59
60      LegalContractVirtualMachine: execute => GlobalSpace
61      assetA: 13.00 stipula_assetA_ed8i9wk, changed: true
62      amountAssetA: 14.00, changed: false
63      assetB: 10.00 stipula_assetB_pl1n5cc, changed: true
64      amountAssetB: 11.00, changed: false
65      Bob: f3hVW1Amltnqe3KvOTO0eT7AU23FAUKdgmCluZB+nss=
        ↪  MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDErzzgD2ZslZxciFAiX3/ot7l
        ↪  rkZDw4148jFZrsDZPE6CVs9xXFSHGgy/mFvIFLXhnChO6Nyd2be3lbgeavLMCMV
        ↪  UiTStXr117Km17keWpb3sItkKKsLFBOcIIU8XXowI/OhzQN2XPZYESHgjdQ5vwE
        ↪  j2YyueiS7WKP94YWz/pswIDAQAB, changed:
        ↪  false
```

```
66      Alice: ubL35Am7TimL5R4oMwm2OxgAYA3XT3BeeDE56oxqdLc=
        ↪ MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCo/GjVKS+3gAA55+kko41yINd⌋
        ↪ OcCLQMSBQyuTTkKHE1mhu/TgOpivM0wLPsSga8hQMr3+v3aR0IF/vfCRf6SdiXm⌋
        ↪ Wx/jflmEXtnT6fkGcnV6dGNUpHWXSpwUIDt0N88jfnEqekx4S+KDCKg99sGEeHe⌋
        ↪ T65fKS8lB0gjHMt9AOriwIDAQAB, changed:
        ↪ false
67      waitTimeBeforeSwapping: 100, changed: false
68
69      LegalContractVirtualMachine: execute => The argument space is empty
70
71      LegalContractVirtualMachine: execute => The data space is empty
72
73      Global state of the execution
74      running -> false
75      executionPointer -> 10
76      executionPointer (with offset) -> 74
77      length of the program -> 11
78      length of the program (with offset) -> 75
79      VirtualMachine: Updating the global store...
80      VirtualMachine: Global store updated
81      VirtualMachine: Ready to dequeue a value...
82      VirtualMachine: I'm waiting...
```

The description of the following flow is illustrated in figure 4.6. When the `EventTrigger` added the request to the request queue, the virtual machine will dequeue the request (**2**) and execute the function that encodes the obligation. If the conditions exist to execute the obligation, then the virtual machine will execute the function (**3**) and will send the processing result to the *Storage* module (**4**). If there are no conditions to perform the obligation, the virtual machine will not perform the function.

4.8.3 Bike rental

The context of use of this agreement has been described above (see section 2.4.1). However, since in the current implementation it is not possible to send messages to the contract participants, the illustrated bytecode will refer to the code of the contract *Stipula* of the appendix A.3.1. In the same appendix there is also the complete contract written in *Stipula bytecode*.

Agreement

This first part of the contract defines the variables for:

1. The asset that the `Borrower` he will have to deposit in order to then be able to pay the `Lender` (line 2);
2. In line 3 the following are defined:
   (a) `cost`: the amount of assets that the `Borrower` will have to deposit;
   (b) `rentingTime`: the time available for which the `Borrower` will be able to use the service;
   (c) `use_code`: represents the bicycle code. This code must be provided by the `Lender`;
3. The initial state of the contract state machine (line 4).

```
1   stipula BikeRental {
2     asset wallet:stipula_coin_asd345
3     field cost, rentingTime, use_code
4     init Inactive
```

When the *agreement* function call is made, the contract parties have found an agreement regarding the cost of the service and the duration of the same.

```
6   agreement (Lender, Borrower)(cost, rentingTime){
7         Lender, Borrower: cost, rentingTime
8     } ==> @Inactive
```

The following bytecode is associated with this function in *Stipula*:

```
1   fn agreement Lender,Borrower Inactive real,time
2   global:
3   GINST party Lender
4   GINST party Borrower
5   GINST asset wallet 2 stipula_coin_asd345
6   GINST real cost 2
7   GINST time rentingTime
8   GINST * use_code
9   args:
10  PUSH party :Lender
11  GSTORE Lender
12  PUSH party :Borrower
13  GSTORE Borrower
14  PUSH real :cost
15  GSTORE cost
16  PUSH time :rentingTime
17  GSTORE rentingTime
18  start:
19  end:
20  HALT
```

The structure of the code is very similar to that illustrated for the previous examples. The peculiarity that can be noticed is found in line 8. The `*` symbol means that the compiler was unable to determine the type of the `use_code` variable. Therefore, the type of the variable will have to be defined later at runtime.

The Lender sends the bicycle code

This piece of code allows the `Lender` to send the code of the bicycle to be used by the `Borrower`. In particular, in line 11 it is possible to notice that the code provided by the `Lender`, through the parameter `z`, is stored in the global variable `use_code`.

```
10  @Inactive Lender : offer(z)[] {
11    z -> use_code;
12    _
13  } ==> @Proposal
```

The following bytecode is associated with this function in *Stipula*:

```
21  fn Inactive Lender offer Proposal *
22  args:
23  PUSH * :z
24  AINST * :z
25  ASTORE z
26  start:
27  ALOAD z
28  GSTORE use_code
29  end:
30  HALT
```

In the function signature it is possible to see that an argument of any type (except `asset`) is accepted. In fact, it will be the `Lender` function call to value the variable and to define its type.

The Borrower deposits the funds in the instance of the contract

The code of this function allows the `Borrower` to deposit funds (lines 15 and 16) and to schedule an event that will execute the obligation at time `now + rentingTime` (line 17 to line 20). The code that encodes the obligation will send the funds contained in the contract instance to the `Lender`.

```
14  @Proposal Borrower : accept()[y]
15        (y == cost) {
16            y -o wallet;
17            now + rentingTime >>
18                @Using {
19                    wallet -o Lender
20                } ==> @End
21  } ==> @Using
```

The following bytecode is associated with this function in *Stipula*:

```
31  fn Proposal Borrower accept Using asset
32  args:
33  PUSH asset :y
34  AINST asset :y
35  ASTORE y
36  start:
37  ALOAD y
38  GLOAD cost
39  ISEQ
40  JMPIF if_branch
41  RAISE AMOUNT_NOT_EQUAL
42  JMP end
43  if_branch:
44  ALOAD y
45  GLOAD wallet
46  DEPOSIT wallet
47  GLOAD rentingTime
```

```
48    PUSH time now
49    ADD
50    TRIGGER obligation_1
51    end:
52    HALT
```

The code is very similar to the example we did earlier for the evented asset swap. The code that implements the obligation is defined as follows:

```
61    obligation Using obligation_1 End
62    start:
63    PUSH real 100 2
64    GLOAD wallet
65    GLOAD Lender
66    WITHDRAW wallet
67    end:
68    HALT
```

End of the contract

If the event to execute the obligation has not yet been triggered, the `Borrower` can terminate the contract by calling the `end` function. Calling this function sends the funds contained in the contract instance to the `Lender`.

```
22    @Using Borrower : end()[] {
23          wallet -o Lender;
24          _
25    } ==> @End
```

The following bytecode is associated with this function in *Stipula*:

```
53    fn Using Borrower end End
54    start:
55    PUSH real 100 2
56    GLOAD wallet
57    GLOAD Lender
58    WITHDRAW wallet
59    end:
60    HALT
```

The code that implements this function, both of the *Stipula* contract and of the bytecode, is almost similar to the code that encodes the obligation, illustrated above.

Example of execution

An example of execution of this contract is illustrated in the appendix A.3.2. Only a few steps will be shown in this section. For simplicity, the value for `rentingTime` is equal to 100, that is, after the `Borrower` has deposited its asset, it will take 100 seconds before the `Borrower` terms of using the service.

Two executions were made for this contract:

1. On the first run, the `Borrower` calls the `end` function before the event that executes the obligation code is triggered;

2. In the second execution, instead, the event that executes the obligation code is triggered first and then the `Borrower` calls the `end` function, which will fail.

Most of the steps of the two executions are the same, the steps that differ according to the execution are specifically indicated.

offer call Request to the server to make the `offer` function call:

```
{
  "message": {
    "contractInstanceId": "4cc1f3c7-5cb4-4528-b15a-8e5cacf5b18a",
    "functionName": "offer",
    "arguments": [
      {
        "argument": {
          "first": "real",
          "second": "z",
          "third": "100 2"
        }
      }
    ],
    "type": "FunctionCall"
  },
  "signatures": {
    "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCo/GjVKS+3gAA55+kko41yIN
    ↪ dOcCLQMSBQyuTTkKHE1mhu/TgOpivM0wLPsSga8hQMr3+v3aR0IF/vfCRf6
    ↪ SdiXmWx/jflmEXtnT6fkGcnV6dGNUpHWXSpwUIDt0N88jfnEqekx4S+KDCK
    ↪ g99sGEeHeT65fKS81B0gjHMt9AOriwIDAQAB":
    ↪ "bJ0xnIhcDlPMmKYx7h8jjX8Q7PaSdqxRg7xq/zTM0vEKqJVDIN0JcT8Qj7
    ↪ jEX5Pwm2YOq+kSwEAxqlPzwoZoQNhe6FPyz6dbj9/LQ0rg79x4QD5ZrCawp
    ↪ cbbtJ/U5l1RPGvl06EdHeQc4YFlsIW4yywD1XlKtfJc7IJwes/iKrE="
  }
}
```

Through this function call, the global variable `use_code` henceforth it will be of type `real` (see section 4.8.3).

Server response:

```
{
  "statusCode": 200,
  "statusMessage": "Success",
  "type": "SuccessDataResponse"
}
```

accept call Request to the server to make the `accept` function call:

```
{
  "message": {
    "contractInstanceId": "4cc1f3c7-5cb4-4528-b15a-8e5cacf5b18a",
    "functionName": "accept",
    "arguments": [
      {
        "argument": {
          "first": "asset",
          "second": "y",
          "third": {
            "ownershipId": "1ce080e5-8c81-48d1-b732-006fa1cc4e2e",
```

```
                "address":
                ↪  "f3hVW1Amltnqe3KvOT00eT7AU23FAUKdgmCluZB+nss=",
                "unlockScript": "PUSH str CJ3CdFnd6QiRoNaxxJN6sEYkmhKsS
                ↪  Ki0SP5YXiSGhygZs+EMyE2bPrI+hRL4PSA0vLh0X6PNpDhTaPxx
                ↪  4kc1LEk9su8+6kkDvi3xpLG9bDoPjss+LEPXUjPTcGVB/3jITb8
                ↪  W+GmX1kDYhGHKtSuhvxBjTwwbtok4gRDD1BcMX/o=\nPUSH str
                ↪  MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDErzzgD2ZslZx
                ↪  ciFAiX3/ot7lrkZDw4148jFZrsDZPE6CVs9xXFSHGgy/mFvIFLX
                ↪  hnChO6Nyd2be3lbgeavLMCMVUiTStXr117Km17keWpb3sItkKKs
                ↪  LFBOcIIU8XXowI/OhzQN2XPZYESHgjdQ5vwEj2YyueiS7WKP94Y
                ↪  Wz/pswIDAQAB\n"
              }
            }
          }
        ],
        "type": "FunctionCall"
      },
      "signatures": {
        "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDErzzgD2ZslZxciFAiX3/ot7
        ↪  lrkZDw4148jFZrsDZPE6CVs9xXFSHGgy/mFvIFLXhnChO6Nyd2be3lbgeav
        ↪  LMCMVUiTStXr117Km17keWpb3sItkKKsLFBOcIIU8XXowI/OhzQN2XPZYES
        ↪  HgjdQ5vwEj2YyueiS7WKP94YWz/pswIDAQAB":
        ↪  "wL3r61IgwBGau7S7V967ZSA8B0lLiOMi0qai1YGQVFXnCTvL9WDVMGTwp7
        ↪  XXAQ77f23Hw5y6Ho5SFUMRRfaTLguIJBx9twRSUfpTP4bh3K4RB2yg32rkO
        ↪  P16G2vIfEirTT+v2wmp1f10pY+dY/QdMzua7EFdQNmL7PhJnA96CpM="
      }
    }
```

Server response:

```
    {
      "statusCode": 200,
      "statusMessage": "Success",
      "type": "SuccessDataResponse"
    }
```

Version 1 - end call Request to the server to make the `end` function call:

```
    {
      "message": {
        "contractInstanceId": "4cc1f3c7-5cb4-4528-b15a-8e5cacf5b18a",
        "functionName": "end",
        "arguments": [],
        "type": "FunctionCall"
      },
      "signatures": {
        "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDErzzgD2ZslZxciFAiX3/ot7
        ↪  lrkZDw4148jFZrsDZPE6CVs9xXFSHGgy/mFvIFLXhnChO6Nyd2be3lbgeav
        ↪  LMCMVUiTStXr117Km17keWpb3sItkKKsLFBOcIIU8XXowI/OhzQN2XPZYES
        ↪  HgjdQ5vwEj2YyueiS7WKP94YWz/pswIDAQAB":
        ↪  "hU6i0eGRNcZB+ZCxeLCPBM31iai412yczQ4/Td+roq9jnBU7agWfuOyVl/
        ↪  6fCKdTZcKkxASJs1tCpe4bLlpUHt01lFlGM8n9+sPHXl+1/jXMngmmPhuNU
        ↪  PrtsD7PGeFtuC3JJkcqTq3WkyWz6nVdn55bzX6BxleN/I6MPgmDroc="
      }
    }
```

Server response:

```
1       {
2         "statusCode": 200,
3         "statusMessage": "Success",
4         "type": "SuccessDataResponse"
5       }
```

Version 1 - Trigger of the event and non-execution of the obligation This situation occurs when the customer, who has used the service, has returned the bicycle before the end of use of the service.

From the server logs it can be seen that the event was triggered and the code encoding the obligation was not executed:

```
1       EventTrigger: A new scheduled request has been triggered =>
        ↪  EventTriggerSchedulingRequest{
2         request=CreateEventRequest{
3           obligationFunctionName='obligation_1',
4           time=1680036610
5         },
6         contractId='622ad60b-ab1f-4c2c-9f64-1307c046b55d',
7         contractInstanceId='4cc1f3c7-5cb4-4528-b15a-8e5cacf5b18a'
8       }
9       EventTrigger: Enqueuing the request...
10      EventTrigger: Notifying the virtual machine...
11      EventTrigger: Virtual machine notified
12      EventTrigger: Removing the request from EventTriggerHandler...
13      VirtualMachine: Ready to dequeue a value...
14      VirtualMachine: Request received => Pair{
15        first=null,
16        second=EventTriggerSchedulingRequest{
17          request=CreateEventRequest{
18            obligationFunctionName='obligation_1',
19            time=1680036610
20          },
21          contractId='622ad60b-ab1f-4c2c-9f64-1307c046b55d',
22          contractInstanceId='4cc1f3c7-5cb4-4528-b15a-8e5cacf5b18a'
23        }
24      }
25      VirtualMachine: Just received a trigger request
26      VirtualMachine: This function cannot be called in the current state
27      VirtualMachine: Obligation function name => obligation_1
28      VirtualMachine: Current state => DfaState{name='End'}
29      VirtualMachine: Next state => null
30      VirtualMachine: Ready to dequeue a value...
31      VirtualMachine: I'm waiting...
```

From line 20 to line 22 it can be seen that the obligation has not been performed because the current state of the contract instance is `@End`. Instead, the state in which the obligation should be executed is `@Using`. For this reason it was not possible to fulfill the obligation.

Version 2 - Event trigger and execution of the obligation This situation occurs when the customer, who has used the service, has not returned the bicycle before the end of use of the service.

From the server logs it can be seen that the event was triggered, the code encoding the obligation was loaded and executed:

```
EventTrigger: A new scheduled request has been triggered => EventTriggerSchedulingRequest{
  request=CreateEventRequest{
    obligationFunctionName='obligation_1',
    time=1680037664
  },
  contractId='51d909ae-45f8-47d2-90de-40699c8a8a3d',
  contractInstanceId='1a7c6469-b4a3-4c67-8a43-ca60514345f6'
}
EventTrigger: Enqueuing the request...
EventTrigger: Notifying the virtual machine...
EventTrigger: Virtual machine notified
EventTrigger: Removing the request from EventTriggerHandler...
VirtualMachine: Ready to dequeue a value...
VirtualMachine: Request received => Pair{
  first=null,
  second=EventTriggerSchedulingRequest{
    request=CreateEventRequest{
      obligationFunctionName='obligation_1',
      time=1680037664
    },
    contractId='51d909ae-45f8-47d2-90de-40699c8a8a3d',
    contractInstanceId='1a7c6469-b4a3-4c67-8a43-ca60514345f6'
  }
}
VirtualMachine: Just received a trigger request
loadObligationFunction: Loading the obligation function...
loadObligationFunction: Obligation function loaded
VirtualMachine: Function
start:
PUSH real 100 2
GLOAD wallet
GLOAD Lender
WITHDRAW wallet
end:
HALT

loadBytecode: Loading the bytecode...
loadBytecode: Bytecode loaded

VirtualMachine: loadBytecode
start:
PUSH real 100 2
GLOAD wallet
GLOAD Lender
WITHDRAW wallet
end:
HALT

LegalContractVirtualMachine: execute => Final state of the execution below
LegalContractVirtualMachine: execute => The stack is empty

LegalContractVirtualMachine: execute => GlobalSpace
```

```
53      rentingTime: 100, changed: false
54      wallet: 11.00 stipula_coin_asd345, changed: true
55      cost: 12.00, changed: false
56      Borrower: f3hVW1Amltnqe3KvOT00eT7AU23FAUKdgmCluZB+nss=
        ↪  MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDErzzgD2ZslZxciFAiX3/ot7l
        ↪  rkZDw4148jFZrsDZPE6CVs9xXFSHGgy/mFvIFLXhnChO6Nyd2be3lbgeavLMCMV
        ↪  UiTStXr117Km17keWpb3sItkKKsLFBOcIIU8XXowI/OhzQN2XPZYESHgjdQ5vwE
        ↪  j2YyueiS7WKP94YWz/pswIDAQAB, changed:
        ↪  false
57      use_code: 1.00, changed: false
58      Lender: ubL35Am7TimL5R4oMwm2OxgAYA3XT3BeeDE56oxqdLc=
        ↪  MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCo/GjVKS+3gAA55+kko41yINd
        ↪  OcCLQMSBQyuTTkKHE1mhu/TgOpivMOwLPsSga8hQMr3+v3aROIF/vfCRf6SdiXm
        ↪  Wx/jflmEXtnT6fkGcnV6dGNUpHWXSpwUIDt0N88jfnEqekx4S+KDCKg99sGEeHe
        ↪  T65fKS8lBOgjHMt9AOriwIDAQAB, changed:
        ↪  false
59
60      LegalContractVirtualMachine: execute => The argument space is empty
61
62      LegalContractVirtualMachine: execute => The data space is empty
63
64      Global state of the execution
65      running -> false
66      executionPointer -> 6
67      executionPointer (with offset) -> 67
68      length of the program -> 7
69      length of the program (with offset) -> 68
70      VirtualMachine: Updating the global store...
71      VirtualMachine: Global store updated
72      VirtualMachine: Ready to dequeue a value...
73      VirtualMachine: I'm waiting...
```

In this case, however, it was possible to execute the code that encodes the obligation because the current state of the contract instance is `@Using` and coincides with the state in which the obligation is to be performed.

Version 2 - end call Here we show the example in which the user tries to call the `end` function, to notify the company of the end of using the service. However, the call to this function took place after the maximum time established by the contract, and therefore the penalty foreseen by the contract was activated.

Request to the server to make the `end` function call:

```
1       {
2         "message": {
3           "contractInstanceId": "1a7c6469-b4a3-4c67-8a43-ca60514345f6",
4           "functionName": "end",
5           "arguments": [],
6           "type": "FunctionCall"
7         },
8         "signatures": {
```

```
          "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDErzzgD2ZslZxciFAiX3/ot7
          ↪ lrkZDw4148jFZrsDZPE6CVs9xXFSHGgy/mFvIFLXhnChO6Nyd2be3lbgeav
          ↪ LMCMVUiTStXr117Km17keWpb3sItkKKsLFBOcIIU8XXowI/OhzQN2XPZYES
          ↪ HgjdQ5vwEj2YyueiS7WKP94YWz/pswIDAQAB":
          ↪ "Ow8gS8d5SChD3E5CgtsnFHTRWskdeWW2IsTLQJk92mS40LfVtPcxuDiIbz
          ↪ L7xWwtUTMFxza+/TSxU+rMsVvMqQLLyUQ4e6UrLO25+Nr7p5x013JGIaxc1
          ↪ 8G5kqEuS4iEyiqN1479E4ElLROE+VpI5DBAKMegw0h9m5cbtHFN/fA="
        }
      }
```

Server response:

```
      {
        "data": "This function cannot be called in the current state",
        "statusCode": 404,
        "statusMessage": "Error",
        "type": "ErrorDataResponse"
      }
```

4.9 Project management

In this short section we will explain the methods to be able to install an instance of the *Stipula* implementation and how the project was managed.

4.9.1 Pipeline

A *pipeline* is a feature offered by a platform that allows for the automation of various tasks in the software development process. In essence, it is a script that can be triggered by events, such as a new commit in a repository or the opening of a *merge request*. The script can then perform a variety of tasks, such as running tests, authoring code, and deploying applications.

Two pipelines were created for this project:

1. `run-tests.yml`: it is invoked every time the branch `test` or `master`, or, when a *merge request* is closed. This pipeline allows you to run the tests that have been defined in the project. There is currently only one sample test. The goal is to set up a testing environment for future developments;

2. `create-and-push-docker-image.yml`: this pipeline is called every time a new *tag* is created. This pipeline allows you to create a *Docker* image and publish it in a dedicated page of the project (see Zanardo, 2023g and Zanardo, 2023a).

These pipelines lay the groundwork for creating a more sophisticated test environment and for facilitating the installation of an instance of the *Stipula* implementation. In the appendix C there is the code of both pipelines.

4.9.2 Issues, milestones ans releases

To track the evolution of the project and the problems that arise during development, the *issue* provided by GitHub was used (Zanardo, 2023d). To better organize the work, these issues have been collected in *milestone* (2023f). The completion of all the issues of a milestone leads to the creation of a *deliverable* (Zanardo, 2023h), that is, a working version of the project that brings new features. Each deliverable is specified

by a *release* (note the image 4.10), which informs all the features introduced and any problems solved. In the milestones page it is possible to notice that the work has been mainly organized in *versions* (note the image 4.11). It can also be noted that the work has been geared towards future versions (note the image 4.12). The current version is `v0.4.2` (Zanardo, 2023e).

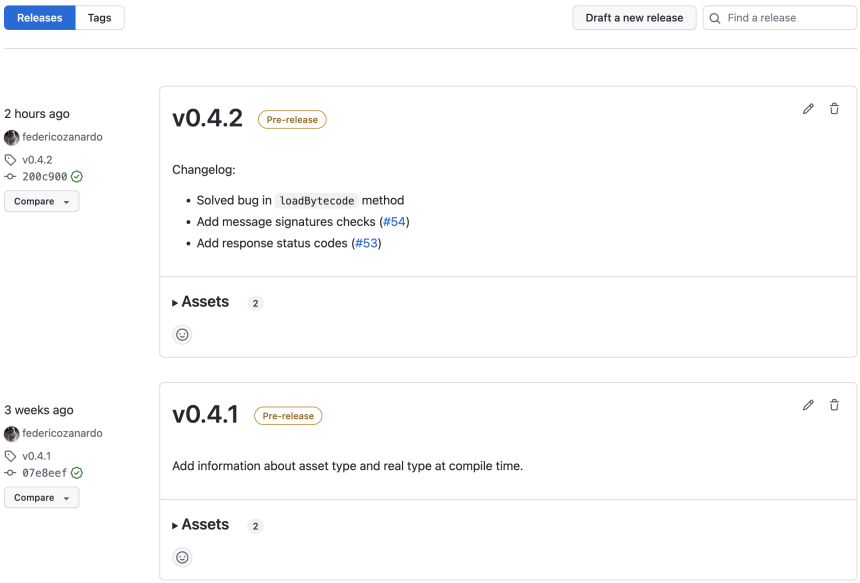


Figure 4.10: Various project releases. For each release it is possible to download the code and start an instance of *Stipula*.

4.9.3 Installation

Starting a *Stipula* server can be done by downloading the code and installing all the packages, or by downloading a Docker image and running the container. For manual installation, you need:

1. Java SDK 8
2. Gradle 7.6.0
3. Gson 2.10.1
4. LevelDb 0.9
5. ANTLR 4.10
6. JUnit 5.8.1

The D appendix contains the code of the `build.gradle` file, which allows you to install and manage the project packages.

For a faster and easier to manage installation it is better to use a Docker image. You can create an instance of *Stipula* with Docker in two ways:

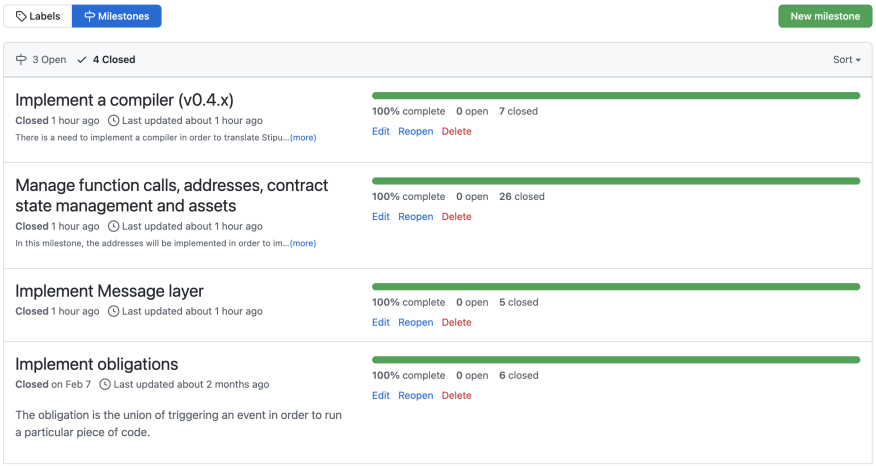


Figure 4.11: Milestones completed.

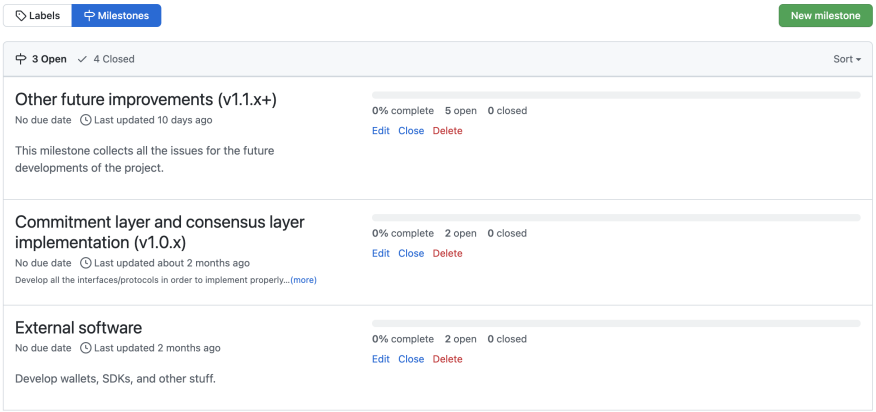


Figure 4.12: Milestone opened.

1. Download the project and run `docker build -t stipula-node:<version> .`, where you must specify the version you want to use instead of `<version>`. You also specify the version in the `docker-compose.yml` file and execute the command `docker-compose -f docker-compose.yml up -d`;

2. Download the Docker files and specify in the `docker-compose.yml` file that the image you want to use must be downloaded from a particular page (Zanardo, 2023a), that is, `image: "ghcr.io/federicozanardo/stipula-node:<version>"`. To start the container, use the command `docker-compose -f docker-compose.yml up -d`.

The benefit you get is that you can run an instance on any machine that supports Docker and takes the responsibility off managing package updates.

Furthermore, each Docker image available at the address

```
ghcr.io/federicozanardo/stipula-node:<version>
```

is an image that is created at each new release, to which it is obviously subjected to tests. In the appendix E there is the code of the `Dockerfile` and `docker-compose.yml`.

Due to the limitations of the implemented architecture, to carry out tests and demonstrate the functioning of the implemented implementation, there is the need to initialize assets and single-use-seals for addresses. To do this, it is necessary to set the `SEED` environment variable: by setting `yes`, assets and single-use-seals will be created when the program is started; valuing with `no`, this procedure will not be performed. The enhancement of this environment variable can be done inside the `docker-compose.yml` (line 12). The following chapter will illustrate the limits of this architecture and propose solutions. In the future, this database *seeding* procedure will be removed.

Chapter 5

Missing features and future developments

This chapter will illustrate the missing features, the limitations of the implemented architecture, the possible optimizations that can be implemented and future developments for the project as a whole.

5.1 Missing features

In this section we introduce the missing features for a complete implementation of the *Stipula* language. The reason for the lack of these features is not due to a limitation of the built architecture, but due to a lack of time to implement them.

5.1.1 Language features not implemented in the current version

The language offers several features for writing contracts. Many of these features require certain properties to be guaranteed, which are often complex to maintain. In the architecture illustrated above, all the features of the language have been implemented, except one: the sending of messages from the contract to the customer. The original example of the `BikeRental` contract (see section 2.4.1) foresaw:

1. When the user called the `accept` function, the bicycle code was sent to the customer. In particular, the complete code would have been:

```
14      ...
15      @Proposal Borrower : accept()[y]
16          (y == cost) {
17              y -o wallet;
18              use_code -> Borrower
19              ...
```

2. When the `accept` function is required was executed, the contract would notify the customer of the end of the service, by sending the `"End_Reached"` message. In particular, the complete code would have been:

```
18      ...
19      now + rentingTime >>
20        @Using {
21          "End_Reached" -> Borrower
22          wallet -o Lender
23        } => @End
24      ...
```

The architecture is ready to implement this feature, that is, the infrastructure for communication between the virtual machine and the client has already been implemented. The missing part is figuring out the necessary data structures and response messages to send to the client.

Due to time constraints, it was not possible to implement the *syntactic sugar* required by the language. The language expects the following syntactic sugar:

1. ```
   ...
   @State1,@State2 Party1,Party2 : functionName()[]
   ...
   ```

   That is, allowing a specific function to be called from multiple parties and/or from multiple states. The implementation of this syntactic sugar involves both the compiler and the virtual machine: the compiler must produce optimized bytecode, that is, instead of writing the function body for each state and for each party, one could update the bytecode language to notify the virtual machine that that code can be invoked from different parties and in different states. The benefits would be obtained from the point of view of memory, as it would be possible to save space instead of duplicating the body of the function each time. In a distributed context, this represents an important point, because it is important to minimize memory usage as much as possible. It is necessary to clarify this is a pure consideration from the point of view of optimization and not from the point of view of expressiveness. The lack of this syntactic sugar does not diminish the expressiveness of the language: in fact, it is possible to write the same function code several times in the bytecode with different states and callers. However, the implementation of this syntactic sugar was not possible due to lack of time.

2. See Silvia Crafa and Cosimo Laneve, 2022

   ```
   ...
   ~ @End _ : block(x) {
     x -> _
   } ==> @Exception
   ...
   ```

   Similarly to the previous point, the implementation of this syntactic sugar involves the compiler and the virtual machine. The meaning of this code is as follows: the `block` function can be invoked by any party (`"_"` notation) provided that the duration of the contract has not expired, that is, the contract is not in the `@End` state.

5.1.2 Single-use-seals merge

The current version has an important limitation when the user has to make a payment to an instance of a contract. In order to make a payment, the user must have available
   ```

a single-use-seal of the exact quantity required by the contract: if the user does not have a single-use-seal of the requested quantity, the user cannot make the payment. This problem represents a strong limit to be able to massively use this implementation. One proposed solution is to *merge* single-use-seals. Suppose Alice has to pay the contract `C_1` 15 `StipulaCoin`. Alice has the following single-use-seals available (note the image 5.1):

1. Single-use-seal 1 (`S_1`): 5 `StipulaCoin`;
2. Single-use-seal 2 (`S_2`): 7 `StipulaCoin`;
3. Single-use-seal 3 (`S_3`): 2 `StipulaCoin`;
4. Single-use-seal 4 (`S_4`): 2 `StipulaCoin`;

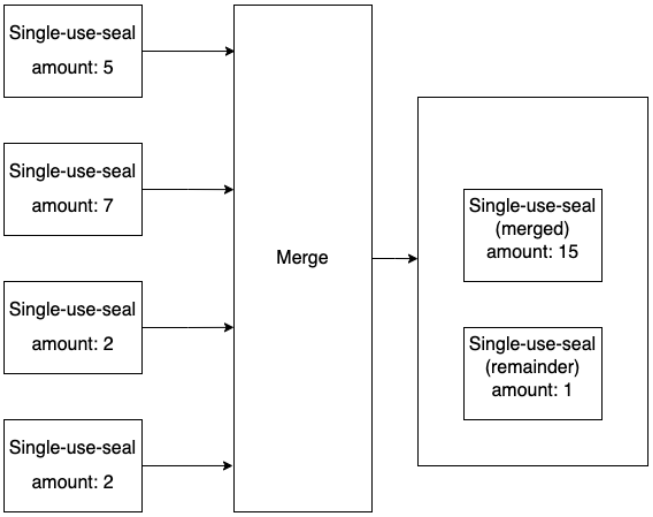


Figure 5.1: Example of single-use-seals merge.

In the current version Alice could not pay the contract as she does not have a single-use-seal of 15 `StipulaCoin`, however the sum of all available funds is 16 `StipulaCoin`, enough to be able to carry out the transaction. The proposed solution consists first of all in modifying the message to make function calls (see 4.3.4), specifically when a *Pay-to-Contract* must be made. The message must be able to collect multiple single-use-seals in its payload. More precisely:

1. `unlockScript` must be provided for each single-use-seals, so that the user proves ownership of the funds;
2. Create new single-use-seals:
   (a) One represents the single-use-seal that will be sent to the contract (`merged`);
   (b) The other single-use-seal represents the *remainder* (`remainder`), that is, the difference between the sum of all the single-use-seals in input minus the amount of asset needed to contract.

I single-use-seals `merged` and `remainder` are the new single-use-seals that will need to be stored. The identifier of these single-use-seals is computed from the hash of the input single-use-seals identifiers. The reason is to decrease the probability of collision between the identifiers of the other single-use-seals. The `merged` must be sent with the contract and therefore `unlockScript` must be provided, to demonstrate possession of the single-use-seal. Instead, you don't need to supply `unlockScript` for the `remainder`, as it is not to be spent in this transaction.

An example is shown below:

1. Line 5 specifies that the payment is a *single-use-seal merge* (`merge`). From line 8 to line 29, all single-use-seals that Alice wants to merge are specified. You can see that for each single-use-seal, Alice has provided cryptographic proof of ownership of those funds. In particular, on line 12, 17, 22, 27 it is possible to note the presence of the `unlockScript` field.

   For `<ownershipId_S_1>` we refer to the identifier of the *first* single-use-seal that Alice wants to spend and for `<unlockScript_S_1>` refers to the `unlockScript` del *first* single-use-seal. The same goes for `<ownershipId_S_2>`, `<ownershipId_S_3>`, `<ownershipId_S_4>`, `<unlockScript_S_2>`, `<unlockScript_S_3>` and `<unlockScript_S_4>`

```
1       ...
2       "arguments": [
3         {
4           "argument": {
5             "first": "merge",
6             "second": "y",
7             "third": {
8               "input": [
9                 {
10                  "ownershipId": "<ownershipId_S_1>",
11                  "address": "<Alice_address>",
12                  "unlockScript": "<unlockScript_S_1>"
13                },
14                {
15                  "ownershipId": "<ownershipId_S_2>",
16                  "address": "<Alice_address>",
17                  "unlockScript": "<unlockScript_S_2>"
18                },
19                {
20                  "ownershipId": "<ownershipId_S_3>",
21                  "address": "<Alice_address>",
22                  "unlockScript": "<unlockScript_S_3>"
23                },
24                {
25                  "ownershipId": "<ownershipId_S_4>",
26                  "address": "<Alice_address>",
27                  "unlockScript": "<unlockScript_S_4>"
28                }
29              ],
```

2. From line 30 to line 56, it is possible to notice what is the output of the merger of the previous single-use-seals (`output`). From line 31 to line 45, you can see the

specification of the new single-use-seal `merged`, which will be sent to the contract instance to make the payment. In fact, this new single-use-seal already comes with the `unlockScript` (line 43). By doing so, the contract will be able to verify whether these funds will actually belong to Alice or not. Instead, from line 46 to line 56, it is possible to notice that a new single-use-seal (`remainder`) is created which represents the difference between the sum of all single-use-seals specified in `input` minus the amount of assets required by the contract instance. No `unlockScript` needs to be supplied for this single-use-seal because it must not be spent on this payment.

For `<hash(ownershipId_S_1)>` means the hash of the identifier of the first input single-use-seal and for `<unlockScript_hash(ownershipId_S_1)>` means the `unlockScript` of this new single-use-seal with identifier `<hash(ownershipId_S_1)>`. The same goes for the single-use-seal `remainder`

```
57          "output": {
58            "merged": {
59              "single_use_seal": {
60                "asset_id": "stipula_coin_345",
61                "amount": {
62                  "value": "1500",
63                  "decimals": "2"
64                }
65              },
66              "ownership": {
67                "ownershipId": "<hash(ownershipId_S_1)>",
68                "address": "<Alice_address>",
69                "unlockScript":
                    ↪  "<unlockScript_hash(ownershipId_S_1)>"
70              }
71            },
72            "remainder": {
73              "single_use_seal": {
74                "asset_id": "stipula_coin_345",
75                "amount": {
76                  "value": "100",
77                  "decimals": "2"
78                }
79              }
80            }
81          }
```

Let's illustrate the complete example:

```
1     ...
2       "arguments": [
3         {
4           "argument": {
5             "first": "merge",
6             "second": "y",
7             "third": {
8               "input": [
```

```
            {
              "ownershipId": "<ownershipId_S_1>",
              "address": "<Alice_address>",
              "unlockScript": "<unlockScript_S_1>"
            },
            {
              "ownershipId": "<ownershipId_S_2>",
              "address": "<Alice_address>",
              "unlockScript": "<unlockScript_S_2>"
            },
            {
              "ownershipId": "<ownershipId_S_3>",
              "address": "<Alice_address>",
              "unlockScript": "<unlockScript_S_3>"
            },
            {
              "ownershipId": "<ownershipId_S_4>",
              "address": "<Alice_address>",
              "unlockScript": "<unlockScript_S_4>"
            }
          ],
          "output": {
            "merged": {
              "single_use_seal": {
                "id": "<hash(S_1)>",
                "asset_id": "stipula_coin_345",
                "amount": {
                  "value": "1500",
                  "decimals": "2"
                }
              },
              "ownership": {
                "ownershipId": "<hash(ownershipId_S_1)>",
                "address": "<Alice_address>",
                "unlockScript":
                 ↪  "<unlockScript_hash(ownershipId_S_1)>"
              }
            },
            "remainder": {
              "single_use_seal": {
                "id": "<hash(S_2)>",
                "asset_id": "stipula_coin_345",
                "amount": {
                  "value": "100",
                  "decimals": "2"
                }
              }
            }
          }
        }
```

```
58            }
59          }
60        ],
61        ...
62      }
63      ...
64    }
```

This solution allows you to make payments even if you don't have single-use-seals of the precise quantity required by the contract. The solution requires updating the message format and carrying out further preliminary checks before executing the contract, namely:

1. Checking the single-use-seals of inputs: in addition to verifying ownership via the *Script*, it is also necessary to check that the sum of the inputs is greater than or equal to the quantity required by the contract;

2. Controlling single-use-seals `merged` and optionally `remainder`: verify that the sum of the quantities of the input single-use-seals is equal to the sum of the quantities of `merged` and optionally `remainder`. Also, verify that the single-use-seal `merged` is equal to the quantity required by the contract;

Passing these checks, the input single-use-seals will be updated in *Storage* as spent. The single-use-seals `merged` will be stored in storage directly as spent, while the single-use-seal `remainder` is stored as unspent. At this point, the virtual machine can continue with the execution of the contract.

5.1.3 Creation of assets and their distribution

One of the missing aspects in the current version is the ability to create additional assets beyond the hard-coded one. As an example, hard-coded assets have been created for the implemented version. One of the peculiarities of the *Stipula* language is that of being able to schedule the execution of certain obligations over time. This functionality could be leveraged to manage the creation, issuance and destruction of assets. The creation of an asset should take place by means of a special contract, separate from the classic contracts seen above. In this special contract, the maximum supply, the fractionability of the asset, the name and an identifier would be defined. However, an asset issuance mechanism could also be defined, such as the *halving* for Bitcoin ((see *Bitcoin mining and halving*, *Bitcoin halving explained* and Antonopoulos, 2017)): using the scheduling of events over time, it is possible to establish that periodically a certain amount of assets is entered into the system. Furthermore, it is also possible to define addresses for the *burn* of the assets. Burning an asset is used to remove liquidity from circulation, thereby decreasing available supply and appreciating the asset. If you want to burn a certain amount of assets, you send it to a specific address where you can only deposit and not withdraw. In this way, it is also possible to monitor and verify the amount of burned assets.

5.2 Optimizations

The current architecture has mainly two bottlenecks:

1. *Virtual Machine*: requests to execute function calls of an instance of a contract or the execution of a time-scheduled event are computed *sequentially*;
2. *Storage*: for simplicity in the realization of the implementation, each request, both for reading and for writing, is managed by a *mutex*, to guarantee exclusive access to the memory by a superior module .

Requests addressed to the compiler and requests addressed to the virtual machine are handled in parallel until one has to interact with the *Storage* module.

Virtual Machine

Requests directed to the virtual machine are done *sequentially*. Unfortunately, this implementation choice represents a bottleneck as regards system performance, in particular, precisely for the virtual machine, which is the most used module during the execution of a contract. However, it is possible to implement optimizations, paying attention, however, to avoid cases of competition. Some possible optimizations for the virtual machine are:

1. *Parallelization of single-use-seals*: this is possible thanks to the UTXO model used in the architecture. The operation of this property has been described 1. For example: Alice has two single-use-seals `S_1` and `S_2` and she wants to make two payments to two different contracts `C_1` and `C_2`. With this optimization, Alice can pay the `C_1` contract with `S_1` and `C_2` with `S_2`. The two requests can be run in parallel because they use separate funds and there is no need to run either request first to update the asset balance, as might be the case in an account-balance-based model. The same dynamic would also occur if Alice had to pay two different instances of the same contract or two instances of two different contracts;
2. *Parallelization of function calls*: if a user is interacting with multiple instances of contracts at the same time, the requests can be executed in parallel. If multiple users are interacting with the same instance of a contract, requests will be fulfilled in order of arrival. As regards the execution of obligations (events scheduled over time), the basic principle does not change: the event can be parallelized if the other requests interact in different contract instances, otherwise the execution of the event has priority over the execution of function calls.

These optimizations can significantly increase the *throughput* of requests, especially in a distributed context. However, it is necessary to update the structure of some modules and to pay particular attention to concurrency.

Storage

Requests received by this module could be parallelized by splitting them into *read* and *write* requests. Requests can come from the compiler, from the virtual machine or directly from the *Message Service*. If the module is only receiving read requests, these requests can be executed in parallel; when a write request is received, it takes priority over read requests. As a further optimization, you could make a specific resource exclusive when a write request is received, and leave free access to other resources.

5.3 Limits of the architecture

5.3.1 Computational and memory resources required

One of the factors to increase the decentralization of a network is to allow the creation of nodes that have a low computational capacity and limited storage capacity. If a ledger were to require large computational capabilities and large amounts of memory, this implies that more expensive machines would be needed. The need for more expensive machines increases the centralization of the network, as the subjects who will be able to buy and maintain these machines will be limited in number. Therefore, in the design phase it is also important to take into account the computational and memory aspect. One of the most expensive operations in a decentralized network is the *synchronization* of the nodes. If the ledger used a lot of memory, this would increase the time it takes to synchronize, thus increasing the possibility of transmission errors. Obviously, this phase would be much more complex and difficult to complete for devices with limited computational and memory capabilities. The current implementation stores various information in the *Storage* module. This could pose a problem for especially distributed networks, if decentralized. During the development of the project we focused on saving all the information necessary to allow the execution of the contracts and for the management of the asset transfers. It would be necessary to analyze whether all the information that is currently stored is actually necessary or, on the other hand, it is possible to omit some information because it is possible to deduce it from other information. Obviously, this balance must be calibrated correctly with the computational cost required to retrieve this information: if it is possible to save a minimal set of information that is currently stored, but the operations necessary to retrieve it require a considerable increase in computational resources, this represents one downside. Furthermore, for simplicity and for a limited amount of time available, the information has been stored according to the data structures of Java objects, as the *LevelDB* library allows information to be stored in bytes. A necessary development is to create information storage interfaces for the various storages (*contracts*, *contract-instances*, *assets* and *ownerships*), in order to allow to implement nodes also in other programming languages.

From a computational point of view, the current version of the language is not Turing-complete, so it is not possible to loop and make function calls from other functions. This design choice limits the expressiveness of the language and therefore also limits the possibility of creating programs that may require significant computational resources. The *Script* language is also not Turing-complete, so it is possible to create programs that require little computational resources. Furthermore, non-Turing-completeness also allows you to avoid unwanted behavior and prevent potential security vulnerabilities. In the previous section (5.2), possible optimizations were introduced to increase request throughput. These optimizations must also take into consideration devices that have limited computational capacity: in this case it is preferable to implement certain optimizations and others not, obtaining a lower throughput for greater decentralization. Obviously, these considerations refer exclusively to decentralized networks.

5.4 Current version security issues

One of the key principles that guided the entire development is security from various points of view. In fact, security can be understood as regards:

1. The assets, that is, guaranteeing that the transfer of assets between users and contracts, and vice versa, takes place without loss of quantity or generation of new quantities from scratch;

2. Prevent a user from double-spending;

3. Ensure that after a contract is executed, the correct state is reached.

The architecture presented above has a security problem regarding the transmission of messages between the server/node and the client. The attack that might occur is a *Man-In-The-Middle* (*MITM*) attack. When a user wants to send a message to the *Stipula* server/node, the message is signed. The benefits of signing are:

1. *Authentication*: the recipient can verify that the message was sent by the declared sender, since only the sender has the private key to sign the message;

2. *Integrity*: the recipient can be certain that the message has not been tampered with during transmission, as any alteration of the message would invalidate the signature;

3. *Non-repudiation*: the sender cannot deny having sent the message once signed, since the signature provides verifiable proof of authorship.

Thus, when the *Stipula* server/node receives a message, even if it is not encrypted, the sender has expressed his intentions of wanting to create a new instance of a contract or make a function call. However, when it is the *Stipula* server/node that has to send response messages to the client, these can be intercepted and tampered with. Indeed, in the server/node there are no cryptographic keys that certify the authenticity and integrity of a message. The problems of having cryptographic keys inside a server/node are:

1. If the cryptographic keys are found by an attacker, the latter can send malicious messages to users;

2. If the keys are lost and a new pair is generated, the problem described in the previous point would arise: the user could not think that the keys have been changed by an attacker and therefore could think that they are receiving malicious messages.

However, in a distributed and centralized context, the use of cryptographic keys in the nodes could be a sufficiently secure solution, as the control of the network belongs to a central body which monitors the traffic. Therefore, nodes with a cryptographic key pair and the central body would be able to cope with similar attacks. The problem arises in networks where there is no central body that governs the network.

You can locate the problem in some response messages:

1. When the server/node has to send the response after the *agreement* phase of a contract. When the agreement phase takes place, the server/node creates a new instance of the contract and assigns it a unique identifier. This identifier will be inserted as the response payload. An attacker could intercept this message

and alter it by inserting an identifier that points to an instance of a malicious contract. By doing so, the user would not notice the change of identifier and would go to accept, and therefore have to respect, the obligations defined in the contract uploaded by the attacker;

2. When server/node sends additional payload in responses. Taking as an example the instruction `use_code -> Borrower`, previously defined in 1, the server/node should add the bicycle code as payload to the response. This message could be intercepted by an attacker, replace the bike code with a fake one, and keep the original bike code. At this point the user would not be able to use the service, as the code is not the original one and if the situation were not resolved before the penalty is triggered, i.e. the trigger of the event previously defined in 2, the contract will send all the money to the company.

We are aware of the issues associated with this architecture, but it was not possible to take steps to mitigate these issues due to lack of time.

5.5 Future improvements

5.5.1 Implementation of the consensus module and communication protocols

This represents the most important module in the distributed context. In this module, it will be necessary to develop communication protocols and algorithms that make it possible to determine consensus regarding the result of the execution of a contract.

The design and development of this module will be very complex as it will have to take into account various security aspects, such as spam. A proposal to try to mitigate the phenomenon of spam is the use of **hashcash** (Adam Back, 2002), a *proof-of-work* algorithm which aims to prevent spam and *DoS* attacks (*Denial-of-Service*), making it more difficult and time-consuming for a sender to send large volumes of requests to a service. The way hashcash works requires the sender to solve a computational puzzle that requires significant computational power to complete. The puzzle is to find a hash value that meets certain criteria, such as having a certain number of leading zeros. The sender has to compute many hash values until it finds one that satisfies the criteria, which takes a lot of computing power and time. Once the sender has solved the puzzle and found a valid hash value, they include this value in the request as proof of work. The recipient can then quickly verify the proof of work by checking the hash value, which allows them to determine whether the sender expended enough computing power to send the request. The power of this mechanism consists in the need for a substantial amount of time and energy to create a proof of work, and at the same time, the verification of the latter can be done instantaneously. The concept of hashcash underpins how Bitcoin proof-of-work works. A similar algorithm adapted to *Stipula* could allow the network of nodes to limit the spam introduced by one or more attackers.

Another useful consensus module component is the creation of a *mempool*. When the virtual machine executes an instance of a contract and the execution is successful, the results produced must first be verified with other nodes to verify that the network (or the majority of the network) agrees with the same results. The network of nodes may be congested and therefore requests may not be served immediately. To free up the virtual machine and run other instances of contracts, it might be useful to implement a queue of requests to submit for consent.

5.5.2 Implementation of the commitment module

This layer allows you to communicate with an underlying layer to do the timestamping and commitment of information. In particular, this module will have to provide common interfaces, in order to make the implementation of *Stipula* independent from the layer that will be used. In fact, if you want to use Ethereum as a commitment layer, you will have to develop specific code that allows you to interact directly with the blockchain, and at the same time respect the common interfaces of the *Stipula* commitment module: in doing so the others modules of the architecture will not change. To do this, the modules that would be involved are:

1. The *Storage* module (see sections 3.3.5 and 4.7): this module will have to store inside the data that will indicate how to find the information saved in the commitment layer;

2. The *Commitment* module (see section 3.3.6): this module will have to interface with the underlying layer to instruct which information will have to be saved;

3. The *Communication protocols* module (see section 3.3.7): communication protocols will be needed between the commitment layer and the *Stipula* implementation for the exchange of information.

Also in this case it is useful to implement a mempool, as the layer could be congested and consequently there could be slowdowns in writing the results obtained from the execution of the contracts.

5.5.3 Fees for performance of a contract

All the smart contracts of various blockchains require *fees* to be paid for their execution, as the computational and memory resources of a distributed network are used. This also happens for layer two, such as *Arbitrum* (see *Arbitrum* and *Arbitrum fees*) and *Optimism* (see *Optimism* and *Optimism fees*) for Ethereum. Fees make possible network attacks (such as *spam*) costly in terms of money and/or resources. Furthermore, fees are a useful tool for *prioritizing* requests: when a blockchain is congested, the network prioritizes those transactions that pay more fees than the others.

The current implementation of *Stipula* does not take this dynamic into consideration: if this version were based on a commitment layer, it is not possible to pay commissions for writing the information. Nor are there any commissions for the network of nodes that execute the *Stipula* contracts. As a future development, it may be necessary to separate the payments to be made to a contract from the commission costs for the execution of the same, both for the network of *Stipula* nodes and for a possible commitment layer. This problem may not arise if *HyperLedger Fabric* nodes are used as commitment layer, which do not include commission costs for writing information in the ledger.

5.5.4 Script language extension

The advantages of using the *Script* language have been described in previous chapters. By extending this language, it is possible to create advanced ways to spend funds, such as authorizing a transaction from multiple users or restricting that a certain amount of assets can only be spent after a certain date. The extensibility of the language makes it possible to satisfy certain needs that could arise in certain contexts, for

example: in a corporate context, it could be useful to carry out a certain expense only with the authorization of several figures, such as directors or managing directors. So, extensibility also allows you to create new, more secure ways to manage your assets. Extensibility can be implemented by adding new instructions to the already existing set (see 4.2) or modifying the current ones, tightening or relaxing the constraints.

5.5.5 Implementation of additional software

In order to incentivize the use of *Stipula* and allow developers to build software on top of its implementation, it is necessary to provide a set of tools and software, such as SDKs. In particular, it is very useful for users to have an application that implements the functions of a *wallet*, that is, a software that allows you to view the balance sheet for each asset, sign transactions, view all sent and received transactions, view the contracts it has approved and other privacy-focused features, such as *coin selection* (5). Another context that requires support software is that of writing contracts. Whoever writes the contracts will be a professional figure in the legal field and therefore it will be necessary to provide tools that allow for the translation of the contractual clauses into *Stipula* code. Therefore, it could be useful to modularize the compiler and the virtual machine to develop tools to be integrated into the IDEs: in doing so, before loading a contract into an instance or a *Stipula* node, the person who will write the contract will be able to check whether the written code will be correct and that respects the expected behaviors. Developing additional tools and software requires interacting only with the *Message Service* module (see 4.3), as all requests and replies go through this module. This facilitates the work of developers, as:

1. They must not interact with other modules, such as the virtual machine, whose tasks are very delicate;

2. Message formats are defined, and therefore it is possible to build tools on top of a *Stipula* server or node, without worrying about messages changing structure. Currently, message formats may change over time until the architecture structure is solid and stable. At that point, message formats will no longer have to change, but new messages can be created for new features.

Chapter 6

Conclusion

The thesis work was mainly divided into two phases: the first research phase, both for the fundamental themes of distributed systems and for programming languages for smart contracts; the second stage of architecture development. The second phase however involved a research activity, aimed however at finding implementation solutions. The first research phase lasted from October to December 2022, while the design and development of the entire architecture lasted from December 2022 until the beginning of April 2023.

6.1 Design considerations

The design took a long time to organize the fundamental concepts of the architecture, such as organizing the components and managing their interactions. In particular, a lot of time was required to design:

1. *Virtual Machine* and *Stipula bytecode*
2. Asset management and *Script* language
3. Distributed context and consent

6.1.1 Virtual Machine and Stipula bytecode

The decision to make the *Stipula* language a compiled language required the creation of a target language for execution, namely, the *Stipula bytecode*. The design of this language took some time to devise the necessary *statements*. The goal was to create a minimal set of instructions that would allow for the implementation of all aspects of the high-level language, trying not to make the set of instructions too large, but to reuse the existing instructions as much as possible. Even the design of the virtual machine was not trivial as the goal was to create a component that would allow for the execution of a contract in isolation from the other components of the architecture.

6.1.2 Asset Management and Script Language

The implementation of a UTXO model is more complex than the account-balance-based model, both from a theoretical point of view and from an implementation point of view. The UTXO model requires you to understand a different approach than classic

balance sheet management. Furthermore, the cryptographic aspect combined with asset management complicates the understanding more. However, this model has important advantages for the application context, such as the possibility of cryptographically verifying the ownership of the funds. By doing so, there can be no contract that could misappropriate your funds. It is always the user who approves an asset transfer to a contract. Connected to the UTXO model, the understanding, design and implementation of the *Script* language was also not trivial. Understanding the usefulness of this language is not trivial, as at first glance it might seem like a useless complication to architecture. Instead, as has been explained in this thesis work, the advantage of the *Script* language is twofold: firstly, it allows to cryptographically demonstrate the ownership of a user's funds; secondly, the extensibility of this language will make it possible to expand the methods of transferring assets, minimizing the architectural modules to be updated.

The idea of reproducing Ethereum/Algorand-style assets and combining the UTXO model was not trivial. Ethereum tokens and Algorand assets use a classic account-balance-based model and therefore their functions for smart contracts also adapt to this model. The use of a UTXO model has overturned the classic interaction between the user and the contracts of Ethereum and Algorand.

6.1.3 Distributed context and consent

The entire research phase and the design phase have always taken into consideration the distributed context. In fact, all the design choices have been made taking into consideration that in the future the current architecture will be placed in a possible distributed system. Then the components and modules needed to adapt the current architecture for a distributed system were also thought of. In the research phase, various problems of distributed systems were analyzed and raised, such as the consensus between the nodes of a network. It is a very interesting topic which would require just as much time to study and propose a solution to be integrated into the current project. Due to lack of time, it was not possible to study these topics further.

6.2 Implementation consideration

The project consists of approximately **15,000 lines of code**, of which approximately *4,500* were generated by the ANTLR tool. The code is divided into **107 classes** and in the repository there is a **graph of the dependencies** between the various classes and the various packages (Zanardo, 2023b). Most of the code is dedicated to the development of the *Virtual Machine*, the *Stipula bytecode* and the *compiler*, as they are the most complex components of the whole architecture. In particular:

1. *Stipula bytecode* and *Virtual Machine* (23 classes and 3121 lines of code):
   (a) *Stipula bytecode*: the implementation of the language required a lot of code as it was necessary to implement all its instructions. For each instruction, the necessary checks must be carried out to avoid unwanted behaviour;
   (b) *Virtual Machine*: this module required a lot of code as it has to handle a very complex flow. Must consider receiving and sending payments (*Pay-to-Contract* and *Pay-to-Party*). In particular, for *Pay-to-Contract* it is necessary to check whether the user is the effective owner of the funds. This module also deals with managing communication with the client and also managing the scheduling of obligations;

2. *Compiler* (28 classes and 6261 lines of code with ANTLR tool, 21 classes and 3007 lines of code without the code generated by ANTLR): this module requires a lot of code (excluding the one generated by ANTLR) to be able to visit the *abstract syntax tree* and then to map the *Stipula* statements into the *Specify bytecode* statement.

6.2.1 Structure of the project

The project repository (Zanardo, 2023c) is organized as follows:

1. `.github/workflows`: contains the pipelines described in the 4.9.1 section and in the C appendix;
2. `examples`: this folder contains example contracts (see the 4.8 section) and example code pieces written in *Stipula bytecode*;
3. `gradle/wrapper`, `build.gradle`, `gradlew`, `gradlew.bat` and `settings.gradle`: files needed for Gradle;
4. `src`: contains the architecture implementation code;
5. `Dockerfile`: it is the file that allows you to create a Docker image (see the appendix E);
6. `README.md`: it is a file that introduces the project, its use, the available features and the installation process;
7. `docker-compose.yml`: is a file that allows you to run a Docker container (see the appendix E);

src

The structure of the `src` folder it's the following:

1. `main/java`: contains the architecture implementation code;
2. `test/java`: currently, contains only an example test. In the future, all architecture tests will be collected.

main/java

The structure of the `main/java` folder it's the following:

1. `compiler`: contains all the code related to the compiler implementation (see section 4.4);
2. `constants`: contains a file containing the *constants* shared between the various modules;
3. `exceptions`: contains classes that implement exceptions for data structures;
4. `lib`: contains the code of some libraries in common with the other modules (see section 4.2);

5. `models`:
   (a) `assets`: contains the code that allows you to implement *assets* (see the 4.1.2 section);
   (b) `contract`: contains the code that implements *contracts* and *contract instances* (see section 4.1.1), *Pay-to-Contract*, `Ownership` (see section 4.3.4) and *single-use-seals* (see section 4.6.3) 4.3.4;
   (c) `dto`:
      i. `requests`: contains the code for implementing the messages defined in the 4.3.4 section;
      ii. `responses`: contains the code for the responses to be sent to the client;
   (d) `party`: contains the code for implementing a *party* (see paragraph 4.5.1);
6. `server`: contains the code implementing the *Message Service* module (see the 4.3 section);
7. `shared`: implements a shared memory area between the virtual machine and the `ClientHandler` (see 2);
8. `storage`: contains the code implementing the *Storage* module (see the 4.7 section);
9. `vm`: contains all the code that implements the *Virtual Machine* module (see the 4.6 section) and the implementation of the *Stipula bytecode* (see the section 4.5);
10. `Main.java`: it is the main file from which it is possible to start the implementation instance;

Appendix A

Examples of contracts and execution of contracts

A.1 Asset swap

A.1.1 Complete code

The complete code of the contract written in *Stipula* is the following:

```
1   stipula SwapAsset {
2     asset assetA:stipula_assetA_ed8i9wk,
      ↪  assetB:stipula_assetB_pl1n5cc
3     field amountAssetA, amountAssetB
4     init Inactive
5
6     agreement (Alice, Bob)(amountAssetA, amountAssetB) {
7         Alice, Bob: amountAssetA, amountAssetB
8     } ==> @Inactive
9
10    @Inactive Alice : depositAssetA()[y]
11        (y == amountAssetA) {
12            y -o assetA;
13            _
14    } ==> @Swap
15
16    @Swap Bob : depositAssetBAndSwap()[y]
17        (y == amountAssetB) {
18            y -o assetB
19            assetB -o Alice
20            assetA -o Bob;
21            _
22    } ==> @End
23  }
```

The complete bytecode produced is:

```
fn agreement Alice,Bob Inactive real,real
global:
GINST party Alice
GINST party Bob
GINST asset assetA 2 stipula_assetA_ed8i9wk
GINST asset assetB 2 stipula_assetB_pl1n5cc
GINST real amountAssetA 2
GINST real amountAssetB 2
args:
PUSH party :Alice
GSTORE Alice
PUSH party :Bob
GSTORE Bob
PUSH real :amountAssetA
GSTORE amountAssetA
PUSH real :amountAssetB
GSTORE amountAssetB
start:
end:
HALT
fn Inactive Alice depositAssetA Swap asset
args:
PUSH asset :y
AINST asset :y
ASTORE y
start:
ALOAD y
GLOAD amountAssetA
ISEQ
JMPIF if_branch
RAISE AMOUNT_NOT_EQUAL
JMP end
if_branch:
ALOAD y
GLOAD assetA
DEPOSIT assetA
end:
HALT
fn Swap Bob depositAssetBAndSwap End asset
args:
PUSH asset :y
AINST asset :y
ASTORE y
start:
ALOAD y
GLOAD amountAssetB
ISEQ
JMPIF if_branch
```

```
49    RAISE AMOUNT_NOT_EQUAL
50    JMP end
51    if_branch:
52    ALOAD y
53    GLOAD assetB
54    DEPOSIT assetB
55    PUSH real 100 2
56    GLOAD assetB
57    GLOAD Alice
58    WITHDRAW assetB
59    PUSH real 100 2
60    GLOAD assetA
61    GLOAD Bob
62    WITHDRAW assetA
63    end:
64    HALT
```

A.2 Asset swap with scheduled event

A.2.1 Complete code

The complete code of the contract written in *Stipula* is the following:

```
1   stipula SwapAssetWithEvent {
2     asset assetA:stipula_assetA_ed8i9wk,
      ↪  assetB:stipula_assetB_pl1n5cc
3     field amountAssetA, amountAssetB, waitTimeBeforeSwapping
4     init Inactive
5
6     agreement (Alice, Bob)(amountAssetA, amountAssetB,
      ↪  waitTimeBeforeSwapping) {
7         Alice, Bob: amountAssetA, amountAssetB,
          ↪  waitTimeBeforeSwapping
8     } ==> @Inactive
9
10    @Inactive Alice : depositAssetA()[y]
11        (y == amountAssetA) {
12            y -o assetA;
13            _
14    } ==> @Deposit
15
16    @Deposit Bob : depositAssetB()[y]
17        (y == amountAssetB) {
18            y -o assetB;
19            now + waitTimeBeforeSwapping >>
20                @Swap {
21                    assetB -o Alice
22                    assetA -o Bob
23                } ==> @End
24    } ==> @Swap
```

```
}
```

The complete bytecode produced is:

```
fn agreement Alice,Bob Inactive real,real,time
global:
GINST party Alice
GINST party Bob
GINST asset assetA 2 stipula_assetA_ed8i9wk
GINST asset assetB 2 stipula_assetB_pl1n5cc
GINST real amountAssetA 2
GINST real amountAssetB 2
GINST time waitTimeBeforeSwapping
args:
PUSH party :Alice
GSTORE Alice
PUSH party :Bob
GSTORE Bob
PUSH real :amountAssetA
GSTORE amountAssetA
PUSH real :amountAssetB
GSTORE amountAssetB
PUSH time :waitTimeBeforeSwapping
GSTORE waitTimeBeforeSwapping
start:
end:
HALT
fn Inactive Alice depositAssetA Deposit asset
args:
PUSH asset :y
AINST asset :y
ASTORE y
start:
ALOAD y
GLOAD amountAssetA
ISEQ
JMPIF if_branch
RAISE AMOUNT_NOT_EQUAL
JMP end
if_branch:
ALOAD y
GLOAD assetA
DEPOSIT assetA
end:
HALT
fn Deposit Bob depositAssetB Swap asset
args:
PUSH asset :y
AINST asset :y
ASTORE y
start:
```

```
48    ALOAD y
49    GLOAD amountAssetB
50    ISEQ
51    JMPIF if_branch
52    RAISE AMOUNT_NOT_EQUAL
53    JMP end
54    if_branch:
55    ALOAD y
56    GLOAD assetB
57    DEPOSIT assetB
58    GLOAD waitTimeBeforeSwapping
59    PUSH time now
60    ADD
61    TRIGGER obligation_1
62    end:
63    HALT
64    obligation Swap obligation_1 End
65    start:
66    PUSH real 100 2
67    GLOAD assetB
68    GLOAD Alice
69    WITHDRAW assetB
70    PUSH real 100 2
71    GLOAD assetA
72    GLOAD Bob
73    WITHDRAW assetA
74    end:
75    HALT
```

A.2.2 Complete example of execution

Deploy contract Request to the server for contract deployment:

```
1     {
2     "message": {
3       "sourceCode": "stipula SwapAsset {\n    asset
        ↪  assetA:stipula_assetA_ed8i9wk,
        ↪  assetB:stipula_assetB_pl1n5cc\n    field amountAssetA,
        ↪  amountAssetB, waitTimeBeforeSwapping\n    init Inactive\n\n
        ↪   agreement (Alice, Bob)(amountAssetA, amountAssetB,
        ↪  waitTimeBeforeSwapping) {\n        Alice, Bob: amountAssetA,
        ↪  amountAssetB, waitTimeBeforeSwapping\n    } ==> @Inactive\n\n
        ↪     @Inactive Alice : depositAssetA()[y]\n        (y ==
        ↪  amountAssetA) {\n            y -o assetA;\n            _\n
        ↪  } ==> @Deposit\n\n    @Deposit Bob : depositAssetB()[y]\n
        ↪     (y == amountAssetB) {\n            y -o assetB;\n
        ↪    now + waitTimeBeforeSwapping >>\n                @Swap {\n
        ↪                    assetB -o Alice\n                    assetA
        ↪  -o Bob\n               } ==> @End\n    } ==> @Swap\n}",
4       "type": "DeployContract"
5     },
6     "signatures": {
```

```
        "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCo/GjVKS+3gAA55+kko41yINdO
        ↪  cCLQMSBQyuTTkKHE1mhu/TgOpivM0wLPsSga8hQMr3+v3aR0IF/vfCRf6SdiX
        ↪  mWx/jflmEXtnT6fkGcnV6dGNUpHWXSpwUIDt0N88jfnEqekx4S+KDCKg99sGE
        ↪  eHeT65fKS8lB0gjHMt9AOriwIDAQAB":
        ↪  "MXw6Xje7jDsbk0Cqfrx6z2pWZiUchw8i9+KsYQ5KPVNic4YQtYYn0Ei64Yul
        ↪  npdNS/jTUxMuJnxW8dOAItDbPeR233731Lh3clnR1xWhRezUBNIF0ZAL2iqVH
        ↪  gaHUYeVNXBaZz1QR+xuj1srSarugnX4LshvZSXGTUUR/U7W4bE="
    }
}
```

Server response:

```
{
  "data": "79caadf1-abbe-418a-a9a2-bd132a6f3e9e",
  "statusCode": 200,
  "statusMessage": "Success",
  "type": "SuccessDataResponse"
}
```

The value in the `data` field indicates the identifier of the deployed contract.

Single-use-seals by Alice Server request to get Alice's single-use-seals:

```
{
  "message": {
    "address": "ubL35Am7TimL5R4oMwm2OxgAYA3XT3BeeDE56oxqdLc=",
    "type": "GetOwnershipsByAddress"
  },
  "signatures": {
    "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCo/GjVKS+3gAA55+kko41yIN
    ↪  dOcCLQMSBQyuTTkKHE1mhu/TgOpivM0wLPsSga8hQMr3+v3aR0IF/vfCRf6
    ↪  SdiXmWx/jflmEXtnT6fkGcnV6dGNUpHWXSpwUIDt0N88jfnEqekx4S+KDCK
    ↪  g99sGEeHeT65fKS8lB0gjHMt9AOriwIDAQAB":
    ↪  "MomZTc63z7PfH35c1dL4tjXebcsW+0Zxl0nP1NQdcUFws98DX+bMWI7L0C
    ↪  6IO5lxvkYve4zdio1Crn97FXvngK4aVfiEZEnHOJ0tstq7uQYGErM3DDAAB
    ↪  qPq8HH5yoKnLST2LWpO0oD8G/VXvIE6qMT5D34W1Ci0q4uh+7y3EcY="
  }
}
```

Server response:

```
{
  "data": "[
      Ownership{
        id='2b4a4614-3bb4-4554-93fe-c034c3ba5a9c',
        singleUseSeal=SingleUseSeal{
          assetId='stipula_assetA_ed8i9wk',
          amount=RealType{
            value=1400,
            decimals=2
          },
          lockScript='DUP\nSHA256\nPUSH str ubL35Am7TimL5R4oMwm2Oxg
          ↪  AYA3XT3BeeDE56oxqdLc=\nEQUAL\nCHECKSIG\nHALT\n'
        },
        unlockScript='',
        contractInstanceId=''
      }
    ]",
```

```
        "statusCode": 200,
        "statusMessage": "Success",
        "type": "SuccessDataResponse"
      }
```

Single-use-seals by Bob Server request to get Bob's single-use-seals:

```
      {
        "message": {
          "address": "f3hVW1Amltnqe3KvOT00eT7AU23FAUKdgmCluZB+nss=",
          "type": "GetOwnershipsByAddress"
        },
        "signatures": {
          "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDErzzgD2ZslZxciFAiX3/ot7
          ↪ lrkZDw4148jFZrsDZPE6CVs9xXFSHGgy/mFvIFLXhnChO6Nyd2be3lbgeav
          ↪ LMCMVUiTStXr117Km17keWpb3sItkKKsLFBOcIIU8XXowI/OhzQN2XPZYES
          ↪ HgjdQ5vwEj2YyueiS7WKP94YWz/pswIDAQAB":
          ↪ "hSNodnUyusffNlv+KNq4605pFvqh91pVspFhTgbmWccE/LKM6h4bedpvTg
          ↪ MHoVDezvA7v2XTzmLG5eL3lOeA6I2xJMH32DcV60IPSoh61oVHnwPQcQHY0
          ↪ 39D4y5VSJ0GMQJKIcTEq3fqIdabg7261xUaegHUnXrcyynh9GpMJxk="
        }
      }
```

Server response:

```
      {
        "data": "[
            Ownership{
              id='7a19f50e-eae9-461d-bd58-9946ea39ccf0',
              singleUseSeal=SingleUseSeal{
                assetId='stipula_assetB_pl1n5cc',
                amount=RealType{
                  value=1100,
                  decimals=2
                },
                lockScript='DUP\nSHA256\nPUSH str f3hVW1Amltnqe3KvOT00eT7
                ↪  AU23FAUKdgmCluZB+nss=\nEQUAL\nCHECKSIG\nHALT\n'
              },
              unlockScript='',
              contractInstanceId=''
            }
          ]",
        "statusCode": 200,
        "statusMessage": "Success",
        "type": "SuccessDataResponse"
      }
```

Agreement Request to the server to make the `agreement` function call:

```
      {
        "message": {
          "contractId": "79caadf1-abbe-418a-a9a2-bd132a6f3e9e",
          "arguments": [
            {
              "argument": {
                "first": "real",
                "second": "amountAssetA",
```

```
              "third": "1400 2"
            }
          },
          {
            "argument": {
              "first": "real",
              "second": "amountAssetB",
              "third": "1100 2"
            }
          },
          {
            "argument": {
              "first": "time",
              "second": "waitTimeBeforeSwapping",
              "third": "100"
            }
          }
        ],
        "parties": {
          "Bob": {
            "address": "f3hVW1Amltnqe3KvOT00eT7AU23FAUKdgmCluZB+nss=",
            "publicKey":
            ↪  "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDErzzgD2ZslZxciF
            ↪  AiX3/ot7lrkZDw4148jFZrsDZPE6CVs9xXFSHGgy/mFvIFLXhnChO6N
            ↪  yd2be3lbgeavLMCMVUiTStXr117Km17keWpb3sItkKKsLFBOcIIU8XX
            ↪  owI/OhzQN2XPZYESHgjdQ5vwEj2YyueiS7WKP94YWz/pswIDAQAB"
          },
          "Alice": {
            "address": "ubL35Am7TimL5R4oMwm2OxgAYA3XT3BeeDE56oxqdLc=",
            "publicKey":
            ↪  "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCo/GjVKS+3gAA55+
            ↪  kko41yINdOcCLQMSBQyuTTkKHE1mhu/TgOpivM0wLPsSga8hQMr3+v3
            ↪  aR0IF/vfCRf6SdiXmWx/jflmEXtnT6fkGcnV6dGNUpHWXSpwUIDt0N8
            ↪  8jfnEqekx4S+KDCKg99sGEeHeT65fKS8lB0gjHMt9AOriwIDAQAB"
          }
        },
        "type": "AgreementCall"
      },
      "signatures": {
        "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDErzzgD2ZslZxciFAiX3/ot7
        ↪  lrkZDw4148jFZrsDZPE6CVs9xXFSHGgy/mFvIFLXhnChO6Nyd2be3lbgeav
        ↪  LMCMVUiTStXr117Km17keWpb3sItkKKsLFBOcIIU8XXowI/OhzQN2XPZYES
        ↪  HgjdQ5vwEj2YyueiS7WKP94YWz/pswIDAQAB":
        ↪  "WrqyzudZAGarLbSlxhYD+Ur6+EqTCFiwqBHEL2IsO5Y23Yxv14O3Uzknr
        ↪  wK41L5LPUgVxR3K75AAZ4n+UcUdDNHlm9KHN7rqpsbe7v3yK2q8Qkk6c4IY
        ↪  NPDRFy3Zw62HH94O7tx8CzcvRfdX4fi+RItf4Fa7hb8Ui/crxDEQN8=",
        "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCo/GjVKS+3gAA55+kko41yIN
        ↪  dOcCLQMSBQyuTTkKHE1mhu/TgOpivM0wLPsSga8hQMr3+v3aR0IF/vfCRf6
        ↪  SdiXmWx/jflmEXtnT6fkGcnV6dGNUpHWXSpwUIDt0N88jfnEqekx4S+KDCK
        ↪  g99sGEeHeT65fKS8lB0gjHMt9AOriwIDAQAB":
        ↪  "o/bdsudfHdR4BBd9EVaGYikksIezSEdwhHELH/f7xRD9g4uokO5g8wHph6
        ↪  LOht5dt9Y+dYt+Qrt+zNZzGUP8a50R7WB2gNz0Jn3zndKnVoBVhsda/zEwI
        ↪  A2pqccP2Sda7zCYiFTfgnmlUZZZfxjtLazBUzDE/vVVFcwtXAHYMXk="
      }
```

```
}
```

For simplicity, the value for `waitTimeBeforeSwapping` is equal to 100, that is, after Bob deposits his asset, they will wait 100 seconds before exchanging assets.

Server response:

```
{
  "data": "48819afd-e28f-4037-82fd-1d073ee1d318",
  "statusCode": 200,
  "statusMessage": "Success",
  "type": "SuccessDataResponse"
}
```

The value in the `data` field indicates the identifier of the created contract instance.

depositAssetA call Request to the server to make the `depositAssetA` function call:

```
{
  "message": {
    "contractInstanceId": "48819afd-e28f-4037-82fd-1d073ee1d318",
    "functionName": "depositAssetA",
    "arguments": [
      {
        "argument": {
          "first": "asset",
          "second": "y",
          "third": {
            "ownershipId": "2b4a4614-3bb4-4554-93fe-c034c3ba5a9c",
            "address":
            ↪  "ubL35Am7TimL5R4oMwm2OxgAYA3XT3BeeDE56oxqdLc=",
            "unlockScript": "PUSH str PLjodnT+m3RNIitQAPBDCsRmJPHCq
            ↪  rwZOY/CPiHFZGnl+DRN6soqxMy3ehTFaUwxBjjf7qfBfvTDq5oB
            ↪  ItTFrtz1Rn5SDS1ybdbkwpKaOXVglNOw7ZEG9bbZ1mo1oA7IAjR
            ↪  iIilzUetCstE5rPZIf9XOXr/RQ5AHkZUn2CztsvA=\nPUSH str
            ↪  MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCo/GjVKS+3gAA
            ↪  55+kko41yINdOcCLQMSBQyuTTkKHE1mhu/TgOpivM0wLPsSga8h
            ↪  QMr3+v3aR0IF/vfCRf6SdiXmWx/jflmEXtnT6fkGcnV6dGNUpHW
            ↪  XSpwUIDt0N88jfnEqekx4S+KDCKg99sGEeHeT65fKS8lB0gjHMt
            ↪  9A0riwIDAQAB\n"
          }
        }
      }
    ],
    "type": "FunctionCall"
  },
  "signatures": {
    "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCo/GjVKS+3gAA55+kko41yIN
    ↪  dOcCLQMSBQyuTTkKHE1mhu/TgOpivM0wLPsSga8hQMr3+v3aR0IF/vfCRf6
    ↪  SdiXmWx/jflmEXtnT6fkGcnV6dGNUpHWXSpwUIDt0N88jfnEqekx4S+KDCK
    ↪  g99sGEeHeT65fKS8lB0gjHMt9A0riwIDAQAB":
    ↪  "mo7rInHGBgsYK0igMBDbcWbRLHF93GnpKGj3NYttddc62CdS5yPg+S7XtH
    ↪  SULj50UCMQZRm3l5uPGtJySlaG31m8tV/JtpTSYNuZLOJdt8ViTMYzHPj0O
    ↪  3tI90R5VzZyyqBV7MkYmKkCK9jBAG3v0V1AqPD8wupXmXjzb5jWi1Q="
  }
}
```

Server response:

```
{
  "statusCode": 200,
  "statusMessage": "Success",
  "type": "SuccessDataResponse"
}
```

depositAssetB call Request to the server to make the `depositAssetBAndSwap` function call:

```
{
  "message": {
    "contractInstanceId": "48819afd-e28f-4037-82fd-1d073ee1d318",
    "functionName": "depositAssetB",
    "arguments": [
      {
        "argument": {
          "first": "asset",
          "second": "y",
          "third": {
            "ownershipId": "7a19f50e-eae9-461d-bd58-9946ea39ccf0",
            "address":
            ↪  "f3hVW1Amltnqe3KvOT00eT7AU23FAUKdgmCluZB+nss=",
            "unlockScript": "PUSH str Q0bPh9lThyrg1slz9AGDJDJh1BecN
            ↪  9SlGCeVe3BqLod+zO7q0wvIy8tLognHNBkR8e8zKo6nWGQ8qZ7e
            ↪  gjOmm5BQsqZzt8xL3gBbR36vgk9J3G9ObiTR2Dd7hMqsqyJnLT3
            ↪  aZUPXGc6RZoM/iUFGJUXhq2T6DStvYNKuAH+Lfow=\nPUSH str
            ↪  MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDErzzgD2ZslZx
            ↪  ciFAiX3/ot7lrkZDw4148jFZrsDZPE6CVs9xXFSHGgy/mFvIFLX
            ↪  hnChO6Nyd2be3lbgeavLMCMVUiTStXr117Km17keWpb3sItkKKs
            ↪  LFBOcIIU8XXowI/OhzQN2XPZYESHgjdQ5vwEj2YyueiS7WKP94Y
            ↪  Wz/pswIDAQAB\n"
          }
        }
      }
    ],
    "type": "FunctionCall"
  },
  "signatures": {
    "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDErzzgD2ZslZxciFAiX3/ot7
    ↪  lrkZDw4148jFZrsDZPE6CVs9xXFSHGgy/mFvIFLXhnChO6Nyd2be3lbgeav
    ↪  LMCMVUiTStXr117Km17keWpb3sItkKKsLFBOcIIU8XXowI/OhzQN2XPZYES
    ↪  HgjdQ5vwEj2YyueiS7WKP94YWz/pswIDAQAB":
    ↪  "VpIlGFopW81rCZ85cuFG+ks5UEhOz4Au8YGAtL7RDH+62l7Va159SCjiyy
    ↪  9y/FBTkYqgId74CC1jJnjfOdMiUy7jasgPa9JiJVUZSo5L/pq3e3pA5LWc4
    ↪  cb/8Yslu+Ax8zPmMzRwCPPu9/5jcowvtkO6NcG1NNdW3Np5vP+M9vM="
  }
}
```

Server response:

```
{
  "statusCode": 200,
  "statusMessage": "Success",
  "type": "SuccessDataResponse"
}
```

Event trigger and execution of the obligation From the server logs it can be seen that the event was triggered, the code encoding the obligation was loaded and executed:

```
EventTrigger: A new scheduled request has been triggered => EventTriggerSchedulingRequest{
  request=CreateEventRequest{
    obligationFunctionName='obligation_1',
    time=1680032647
  },
  contractId='79caadf1-abbe-418a-a9a2-bd132a6f3e9e',
  contractInstanceId='48819afd-e28f-4037-82fd-1d073ee1d318'
}
EventTrigger: Enqueuing the request...
EventTrigger: Notifying the virtual machine...
EventTrigger: Virtual machine notified
EventTrigger: Removing the request from EventTriggerHandler...
VirtualMachine: Ready to dequeue a value...
VirtualMachine: Request received => Pair{
  first=null,
  second=EventTriggerSchedulingRequest{
    request=CreateEventRequest{
      obligationFunctionName='obligation_1',
      time=1680032647
    },
    contractId='79caadf1-abbe-418a-a9a2-bd132a6f3e9e',
    contractInstanceId='48819afd-e28f-4037-82fd-1d073ee1d318'
  }
}
VirtualMachine: Just received a trigger request
loadObligationFunction: Loading the obligation function...
loadObligationFunction: Obligation function loaded
VirtualMachine: Function
start:
PUSH real 100 2
GLOAD assetB
GLOAD Alice
WITHDRAW assetB
PUSH real 100 2
GLOAD assetA
GLOAD Bob
WITHDRAW assetA
end:
HALT

loadBytecode: Loading the bytecode...
loadBytecode: Bytecode loaded

VirtualMachine: loadBytecode
start:
PUSH real 100 2
GLOAD assetB
GLOAD Alice
WITHDRAW assetB
PUSH real 100 2
```

```
51      GLOAD assetA
52      GLOAD Bob
53      WITHDRAW assetA
54      end:
55      HALT
56
57      LegalContractVirtualMachine: execute => Final state of the
        ↪  execution below
58      LegalContractVirtualMachine: execute => The stack is empty
59
60      LegalContractVirtualMachine: execute => GlobalSpace
61      assetA: 13.00 stipula_assetA_ed8i9wk, changed: true
62      amountAssetA: 14.00, changed: false
63      assetB: 10.00 stipula_assetB_pl1n5cc, changed: true
64      amountAssetB: 11.00, changed: false
65      Bob: f3hVW1Amltnqe3KvOT00eT7AU23FAUKdgmCluZB+nss=
        ↪  MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDErzzgD2ZslZxciFAiX3/ot7l
        ↪  rkZDw4148jFZrsDZPE6CVs9xXFSHGgy/mFvIFLXhnChO6Nyd2be3lbgeavLMCMV
        ↪  UiTStXr117Km17keWpb3sItkKKsLFBOcIIU8XXowI/OhzQN2XPZYESHgjdQ5vwE
        ↪  j2YyueiS7WKP94YWz/pswIDAQAB, changed:
        ↪  false
66      Alice: ubL35Am7TimL5R4oMwm2OxgAYA3XT3BeeDE56oxqdLc=
        ↪  MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCo/GjVKS+3gAA55+kko41yINd
        ↪  OcCLQMSBQyuTTkKHE1mhu/TgOpivM0wLPsSga8hQMr3+v3aR0IF/vfCRf6SdiXm
        ↪  Wx/jflmEXtnT6fkGcnV6dGNUpHWXSpwUIDt0N88jfnEqekx4S+KDCKg99sGEeHe
        ↪  T65fKS81B0gjHMt9AOriwIDAQAB, changed:
        ↪  false
67      waitTimeBeforeSwapping: 100, changed: false
68
69      LegalContractVirtualMachine: execute => The argument space is empty
70
71      LegalContractVirtualMachine: execute => The data space is empty
72
73      Global state of the execution
74      running -> false
75      executionPointer -> 10
76      executionPointer (with offset) -> 74
77      length of the program -> 11
78      length of the program (with offset) -> 75
79      VirtualMachine: Updating the global store...
80      VirtualMachine: Global store updated
81      VirtualMachine: Ready to dequeue a value...
82      VirtualMachine: I'm waiting...
```

The description of the following flow is illustrated in figure 4.6. When the `EventTrigger` added the request to the request queue, the virtual machine will dequeue the request (**2**) and execute the function that encodes the obligation. If the conditions exist to execute the obligation, then the virtual machine will execute the function (**3**) and will send the processing result to the *Storage* module (**4**). If there are no conditions to perform the obligation, the virtual machine will not perform the function.

Single-use-seals by Alice Server request to get Alice's single-use-seals:

```
{
  "message": {
    "address": "ubL35Am7TimL5R4oMwm2OxgAYA3XT3BeeDE56oxqdLc=",
    "type": "GetOwnershipsByAddress"
  },
  "signatures": {
    "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCo/GjVKS+3gAA55+kko41yIN
    ↪ dOcCLQMSBQyuTTkKHE1mhu/TgOpivM0wLPsSga8hQMr3+v3aR0IF/vfCRf6
    ↪ SdiXmWx/jflmEXtnT6fkGcnV6dGNUpHWXSpwUIDt0N88jfnEqekx4S+KDCK
    ↪ g99sGEeHeT65fKS8lB0gjHMt9AOriwIDAQAB":
    ↪ "MomZTc63z7PfH35c1dL4tjXebcsW+0Zxl0nP1NQdcUFws98DX+bMWI7L0C
    ↪ 6IO5lxvkYve4zdio1Crn97FXvngK4aVfiEZEnHOJ0tstq7uQYGErM3DDAAB
    ↪ qPq8HH5yoKnLST2LWpO0oD8G/VXvIE6qMT5D34W1Ci0q4uh+7y3EcY="
  }
}
```

Server response:

```
{
  "data": "[
      Ownership{
        id='2b4a4614-3bb4-4554-93fe-c034c3ba5a9c',
        singleUseSeal=SingleUseSeal{
          assetId='stipula_assetA_ed8i9wk',
          amount=RealType{
            value=1400,
            decimals=2
          },
          lockScript='DUP\nSHA256\nPUSH str ubL35Am7TimL5R4oMwm2Oxg
          ↪ AYA3XT3BeeDE56oxqdLc=\nEQUAL\nCHECKSIG\nHALT\n'
        },
        unlockScript='PUSH str PLjodnT+m3RNIitQAPBDCsRmJPHCqrwZOY/C
        ↪ PiHFZGnl+DRN6soqxMy3ehTFaUwxBjjf7qfBfvTDq5oBItTFrtz1Rn5
        ↪ SDS1ybdbkwpKaOXVglNOw7ZEG9bbZ1mo1oA7IAjRiIilzUetCstE5rP
        ↪ ZIf9XOXr/RQ5AHkZUn2CztsvA=\nPUSH str
        ↪ MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCo/GjVKS+3gAA55+k
        ↪ ko41yINdOcCLQMSBQyuTTkKHE1mhu/TgOpivM0wLPsSga8hQMr3+v3a
        ↪ R0IF/vfCRf6SdiXmWx/jflmEXtnT6fkGcnV6dGNUpHWXSpwUIDt0N88
        ↪ jfnEqekx4S+KDCKg99sGEeHeT65fKS8lB0gjHMt9AOriwIDAQAB\n',
        contractInstanceId='48819afd-e28f-4037-82fd-1d073ee1d318'
      },
      Ownership{
        id='a325a1ed-612c-4201-b5ef-0a58ff184509',
        singleUseSeal=SingleUseSeal{
          assetId='stipula_assetB_pl1n5cc',
          amount=RealType{
            value=100,
            decimals=2
          },
          lockScript='DUP\nSHA256\nPUSH str ubL35Am7TimL5R4oMwm2Oxg
          ↪ AYA3XT3BeeDE56oxqdLc=\nEQUAL\nCHECKSIG\nHALT\n'
        },
        unlockScript='',
        contractInstanceId=''
      }
```

```
29          ]",
30        "statusCode": 200,
31        "statusMessage": "Success",
32        "type": "SuccessDataResponse"
33      }
```

It is possible to see that the first single-use-seal has been spent and that's what was deposited in the contract instance. Evidence that the funds have been spent is given by the `unlockScript` field. While, the second single-use-seal represents the asset that was in Bob's possession.

Single-use-seals by Bob Server request to get Bob's single-use-seals:

```
1       {
2         "message": {
3           "address": "f3hVW1Amltnqe3KvOT00eT7AU23FAUKdgmCluZB+nss=",
4           "type": "GetOwnershipsByAddress"
5         },
6         "signatures": {
7           "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDErzzgD2ZslZxciFAiX3/ot7
            ↪  lrkZDw4148jFZrsDZPE6CVs9xXFSHGgy/mFvIFLXhnChO6Nyd2be3lbgeav
            ↪  LMCMVUiTStXr117Km17keWpb3sItkKKsLFBOcIIU8XXowI/OhzQN2XPZYES
            ↪  HgjdQ5vwEj2YyueiS7WKP94YWz/pswIDAQAB":
            ↪  "hSNodnUyusffNlv+KNq4605pFvqh91pVspFhTgbmWccE/LKM6h4bedpvTg
            ↪  MHoVDezvA7v2XTzmLG5eL3lOeA6I2xJMH32DcV60IPSoh61oVHnwPQcQHY0
            ↪  39D4y5VSJ0GMQJKIcTEq3fqIdabg7261xUaegHUnXrcyynh9GpMJxk="
8         }
9       }
```

Server response:

```
1       {
2         "data": "[
3             Ownership{
4               id='7a19f50e-eae9-461d-bd58-9946ea39ccf0',
5               singleUseSeal=SingleUseSeal{
6                 assetId='stipula_assetB_pl1n5cc',
7                 amount=RealType{
8                   value=1100,
9                   decimals=2
10                },
11                lockScript='DUP\nSHA256\nPUSH str f3hVW1Amltnqe3KvOT00eT7
                  ↪  AU23FAUKdgmCluZB+nss=\nEQUAL\nCHECKSIG\nHALT\n'
12              },
13              unlockScript='PUSH str Q0bPh9lThyrg1slz9AGDJDJh1BecN9SlGCeV
                ↪  e3BqLod+zO7q0wvIy8tLognHNBkR8e8zKo6nWGQ8qZ7egjOmm5BQsqZ
                ↪  zt8xL3gBbR36vgk9J3G9ObiTR2Dd7hMqsqyJnLT3aZUPXGc6RZoM/iU
                ↪  FGJUXhq2T6DStvYNKuAH+Lfow=\nPUSH str
                ↪  MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDErzzgD2ZslZxciFA
                ↪  iX3/ot7lrkZDw4148jFZrsDZPE6CVs9xXFSHGgy/mFvIFLXhnChO6Ny
                ↪  d2be3lbgeavLMCMVUiTStXr117Km17keWpb3sItkKKsLFBOcIIU8XXo
                ↪  wI/OhzQN2XPZYESHgjdQ5vwEj2YyueiS7WKP94YWz/pswIDAQAB\n',
14              contractInstanceId='48819afd-e28f-4037-82fd-1d073ee1d318'
15            },
16            Ownership{
17              id='fa8a1383-3f42-4d7b-ad2a-d3e47ee1eb4b',
18              singleUseSeal=SingleUseSeal{
```

```
19                assetId='stipula_assetA_ed8i9wk',
20                amount=RealType{
21                  value=100,
22                  decimals=2
23                },
24                lockScript='DUP\nSHA256\nPUSH str f3hVW1Amltnqe3KvOT00eT7
                  ↪  AU23FAUKdgmCluZB+nss=\nEQUAL\nCHECKSIG\nHALT\n'
25              },
26              unlockScript='',
27              contractInstanceId=''
28            }
29          ]",
30        "statusCode": 200,
31        "statusMessage": "Success",
32        "type": "SuccessDataResponse"
33      }
```

It is possible to see that the first single-use-seal has been spent and that's what was deposited in the contract instance. Evidence that the funds have been spent is given by the `unlockScript` field. While, the second single-use-seal represents the asset that was in Alice's possession.

A.3 Bike rental

A.3.1 Complete code

The complete code of the contract written in *Stipula* is the following:

```
1     stipula BikeRental {
2       asset wallet:stipula_coin_asd345
3       field cost, rentingTime, use_code
4       init Inactive
5
6       agreement (Lender, Borrower)(cost, rentingTime){
7           Lender, Borrower: cost, rentingTime
8       } ==> @Inactive
9
10      @Inactive Lender : offer(z)[] {
11          z -> use_code;
12          _
13      } ==> @Proposal
14
15      @Proposal Borrower : accept()[y]
16          (y == cost) {
17              y -o wallet;
18              now + rentingTime >>
19                  @Using {
20                      wallet -o Lender
21                  } ==> @End
22      } ==> @Using
23
24      @Using Borrower : end()[] {
```

```
25          wallet -o Lender;
26          _
27      } ==> @End
28  }
```

The complete bytecode produced is:

```
1   fn agreement Lender,Borrower Inactive real,time
2   global:
3   GINST party Lender
4   GINST party Borrower
5   GINST asset wallet 2 stipula_coin_asd345
6   GINST real cost 2
7   GINST time rentingTime
8   GINST * use_code
9   args:
10  PUSH party :Lender
11  GSTORE Lender
12  PUSH party :Borrower
13  GSTORE Borrower
14  PUSH real :cost
15  GSTORE cost
16  PUSH time :rentingTime
17  GSTORE rentingTime
18  start:
19  end:
20  HALT
21  fn Inactive Lender offer Proposal *
22  args:
23  PUSH * :z
24  AINST * :z
25  ASTORE z
26  start:
27  ALOAD z
28  GSTORE use_code
29  end:
30  HALT
31  fn Proposal Borrower accept Using asset
32  args:
33  PUSH asset :y
34  AINST asset :y
35  ASTORE y
36  start:
37  ALOAD y
38  GLOAD cost
39  ISEQ
40  JMPIF if_branch
41  RAISE AMOUNT_NOT_EQUAL
42  JMP end
43  if_branch:
44  ALOAD y
```

```
45    GLOAD wallet
46    DEPOSIT wallet
47    GLOAD rentingTime
48    PUSH time now
49    ADD
50    TRIGGER obligation_1
51    end:
52    HALT
53    fn Using Borrower end End
54    start:
55    PUSH real 100 2
56    GLOAD wallet
57    GLOAD Lender
58    WITHDRAW wallet
59    end:
60    HALT
61    obligation Using obligation_1 End
62    start:
63    PUSH real 100 2
64    GLOAD wallet
65    GLOAD Lender
66    WITHDRAW wallet
67    end:
68    HALT
```

A.3.2 Complete example of execution

Deploy contract Request to the server for contract deployment:

```
1     {
2       "message": {
3         "sourceCode": "stipula BikeRental {\n    asset
          ↪  wallet:stipula_coin_asd345\n    field cost,
          ↪  rentingTime, use_code\n    init Inactive\n\n
          ↪  agreement (Lender, Borrower)(cost, rentingTime){\n
          ↪     Lender, Borrower: cost, rentingTime\n    } ==>
          ↪  @Inactive\n\n    @Inactive Lender : offer(z)[] {\n
          ↪     z -> use_code;\n        _\n    } ==> @Proposal\n\n
          ↪    @Proposal Borrower : accept()[y]\n        (y ==
          ↪  cost) {\n            y -o wallet;\n            now +
          ↪  rentingTime >>\n                @Using {\n
          ↪         wallet -o Lender\n                } ==> @End\n
          ↪    } ==> @Using\n\n    @Using Borrower : end()[] {\n
          ↪      wallet -o Lender;\n        _\n    } ==> @End\n}\n",
4         "type": "DeployContract"
5       },
6       "signatures": {
```

```
        "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCo/GjVKS+3gAA55+kko
        ↪ 41yINdOcCLQMSBQyuTTkKHE1mhu/TgOpivM0wLPsSa8hQMr3+v3aR
        ↪ 0IF/vfCRf6SdiXmWx/jflmEXtnT6fkGcnV6dGNUpHWXSpwUIDt0N88
        ↪ jfnEqekx4S+KDCKg99sGEeHeT65fKS8lB0gjHMt9AOriwIDAQAB":
        ↪ "UTtjlNtPutvql7jAaC0N6HiD1Q+83hborpEvcjvnzbc6lDwwZioGj
        ↪ p2cOKHpC6YXypyqjQZnp1TeagacO9KSZLSZuduBHwiNE20qEgXTCYr
        ↪ 0oB1Sww09AQgI23vEoIHf7V0SzLdkfTC6DMxD2nBcMju/4z6xGbXnj
        ↪ fwR+sqxkZE="
    }
}
```

Server response:

```
{
  "data": "622ad60b-ab1f-4c2c-9f64-1307c046b55d",
  "statusCode": 200,
  "statusMessage": "Success",
  "type": "SuccessDataResponse"
}
```

The value in the `data` field indicates the identifier of the deployed contract.

Single-use-seals by Lender Server request to get `Lender`'s single-use-seals:

```
{
  "message": {
    "address": "ubL35Am7TimL5R4oMwm2OxgAYA3XT3BeeDE56oxqdLc=",
    "type": "GetOwnershipsByAddress"
  },
  "signatures": {
    "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCo/GjVKS+3gAA55+kko
    ↪ 41yINdOcCLQMSBQyuTTkKHE1mhu/TgOpivM0wLPsSa8hQMr3+v3aR
    ↪ 0IF/vfCRf6SdiXmWx/jflmEXtnT6fkGcnV6dGNUpHWXSpwUIDt0N88
    ↪ jfnEqekx4S+KDCKg99sGEeHeT65fKS8lB0gjHMt9AOriwIDAQAB":
    ↪ "MomZTc63z7PfH35c1dL4tjXebcsW+0Zxl0nP1NQdcUFws98DX+bMW
    ↪ I7L0C6IO5lxvkYve4zdio1Crn97FXvngK4aVfiEZEnHOJ0tstq7uQY
    ↪ GErM3DDAABqPq8HH5yoKnLST2LWpO0oD8G/VXvIE6qMT5D34W1Ci0q
    ↪ 4uh+7y3EcY="
  }
}
```

Server response:

```
{
  "data": "[]",
  "statusCode": 200,
  "statusMessage": "Success",
  "type": "SuccessDataResponse"
}
```

It is possible to notice that the `Lender` it has no funds.

Single-use-seals by Borrower Server request to get `Borrower`'s single-use-seals:

```
{
  "message": {
    "address": "f3hVW1Amltnqe3KvOT00eT7AU23FAUKdgmCluZB+nss=",
    "type": "GetOwnershipsByAddress"
  },
  "signatures": {
    "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDErzzgD2ZslZxciFAiX
    ↪ 3/ot7lrkZDw4148jFZrsDZPE6CVs9xXFSHGgy/mFvIFLXhnChO6Nyd
    ↪ 2be3lbgeavLMCMVUiTStXr117Km17keWpb3sItkKKsLFBOcIIU8XXo
    ↪ wI/OhzQN2XPZYESHgjdQ5vwEj2YyueiS7WKP94YWz/pswIDAQAB":
    ↪ "hSNodnUyusffNlv+KNq4605pFvqh91pVspFhTgbmWccE/LKM6h4be
    ↪ dpvTgMHoVDezvA7v2XTzmLG5eL3lOeA6I2xJMH32DcV60IPSoh61oV
    ↪ HnwPQcQHY039D4y5VSJ0GMQJKIcTEq3fqIdabg7261xUaegHUnXrcy
    ↪ ynh9GpMJxk="
  }
}
```

Server response:

```
{
  "data": "[
      Ownership{
        id='1ce080e5-8c81-48d1-b732-006fa1cc4e2e',
        singleUseSeal=SingleUseSeal{
          assetId='stipula_coin_asd345',
          amount=RealType{
            value=1200,
            decimals=2
          },
          lockScript='DUP\nSHA256\nPUSH str
          ↪ f3hVW1Amltnqe3KvOT00eT7AU23FAUKdgmCluZB+nss=\nEQ
          ↪ UAL\nCHECKSIG\nHALT\n'
        },
        unlockScript='',
        contractInstanceId=''
      }
    ]",
  "statusCode": 200,
  "statusMessage": "Success",
  "type": "SuccessDataResponse"
}
```

Agreement Request to the server to make the `agreement` function call:

```
{
  "message": {
    "contractId": "622ad60b-ab1f-4c2c-9f64-1307c046b55d",
    "arguments": [
      {
        "argument": {
```

```
            "first": "real",
            "second": "cost",
            "third": "1200 2"
          }
        },
        {
          "argument": {
            "first": "time",
            "second": "rentingTime",
            "third": "100"
          }
        }
      ],
      "parties": {
        "Borrower": {
          "address":
          ↪  "f3hVW1Amltnqe3KvOT0eT7AU23FAUKdgmCluZB+nss=",
          "publicKey": "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDE
          ↪  rzzgD2ZslZxciFAiX3/ot7lrkZDw4148jFZrsDZPE6CVs9xXFS
          ↪  HGgy/mFvIFLXhnChO6Nyd2be3lbgeavLMCMVUiTStXr117Km17
          ↪  keWpb3sItkKKsLFBOcIIU8XXowI/OhzQN2XPZYESHgjdQ5vwEj
          ↪  2YyueiS7WKP94YWz/pswIDAQAB"
        },
        "Lender": {
          "address":
          ↪  "ubL35Am7TimL5R4oMwm2OxgAYA3XT3BeeDE56oxqdLc=",
          "publicKey": "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCo
          ↪  /GjVKS+3gAA55+kko41yINdOcCLQMSBQyuTTkKHE1mhu/TgOpi
          ↪  vM0wLPsSga8hQMr3+v3aR0IF/vfCRf6SdiXmWx/jflmEXtnT6f
          ↪  kGcnV6dGNUpHWXSpwUIDt0N88jfnEqekx4S+KDCKg99sGEeHeT
          ↪  65fKS8lB0gjHMt9AOriwIDAQAB"
        }
      },
      "type": "AgreementCall"
    },
    "signatures": {
      "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDErzzgD2ZslZxciFAiX
      ↪  3/ot7lrkZDw4148jFZrsDZPE6CVs9xXFSHGgy/mFvIFLXhnChO6Nyd
      ↪  2be3lbgeavLMCMVUiTStXr117Km17keWpb3sItkKKsLFBOcIIU8XXo
      ↪  wI/OhzQN2XPZYESHgjdQ5vwEj2YyueiS7WKP94YWz/pswIDAQAB":
      ↪  "o9hrAXSkpskhxRdS+7vWSi85DhAlYlPt7EWUkYFsLOmo8ZluA0MNc
      ↪  Lksi2FEFs3f5Gsike0nvrVCKKGVHIQLaBsDr9TgHVBEwNV7IqsvBaT
      ↪  uO7GaIndWmaC2T+oVKzzpuO30p5MWx4ukmZ+c3BjZrtSO60qVZdfYX
      ↪  aa9mByBDPM=",
```

```
34          "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCo/GjVKS+3gAA55+kko
             ↪  41yINdOcCLQMSBQyuTTkKHE1mhu/TgOpivM0wLPsSga8hQMr3+v3aR
             ↪  0IF/vfCRf6SdiXmWx/jflmEXtnT6fkGcnV6dGNUpHWXSpwUIDt0N88
             ↪  jfnEqekx4S+KDCKg99sGEeHeT65fKS8lB0gjHMt9AOriwIDAQAB":
             ↪  "ITY3VKqPbsLtAuSk5xabu2v9pVTaGyypMxEzxhLv+JOgoHmdkCkPV
             ↪  8k3JG4efJR/AdqaZrFvJWg8SGFwylWBKMP0/83GvToRdIBkfHg78v6
             ↪  fOzh+xVRtFH0OhPnPHNksKn9EZwidhJEiNyEKYMfT6VADhaFkjdLCX
             ↪  cYqjUUrKzY="
35        }
36      }
```

For simplicity, the value for `rentingTime` is equal to 100, that is, after the `Borrower` has deposited its asset, it will take 100 seconds before the `Borrower` terms of using the service.

Server response:

```
1     {
2       "data": "4cc1f3c7-5cb4-4528-b15a-8e5cacf5b18a",
3       "statusCode": 200,
4       "statusMessage": "Success",
5       "type": "SuccessDataResponse"
6     }
```

The value in the `data` field indicates the identifier of the created contract instance.

offer call Request to the server to make the `offer` function call:

```
1     {
2       "message": {
3         "contractInstanceId":
          ↪  "4cc1f3c7-5cb4-4528-b15a-8e5cacf5b18a",
4         "functionName": "offer",
5         "arguments": [
6           {
7             "argument": {
8               "first": "real",
9               "second": "z",
10              "third": "100 2"
11            }
12          }
13        ],
14        "type": "FunctionCall"
15      },
16      "signatures": {
17        "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCo/GjVKS+3gAA55+kko
           ↪  41yINdOcCLQMSBQyuTTkKHE1mhu/TgOpivM0wLPsSga8hQMr3+v3aR
           ↪  0IF/vfCRf6SdiXmWx/jflmEXtnT6fkGcnV6dGNUpHWXSpwUIDt0N88
           ↪  jfnEqekx4S+KDCKg99sGEeHeT65fKS8lB0gjHMt9AOriwIDAQAB":
           ↪  "bJ0xnIhcDlPMmKYx7h8jjX8Q7PaSdqxRg7xq/zTM0vEKqJVDIN0Jc
           ↪  T8Qj7jEX5Pwm2YOq+kSwEAxqlPzwoZoQNhe6FPyz6dbj9/LQ0rg79x
           ↪  4QD5ZrCawpcbbtJ/U5l1RPGvl06EdHeQc4YFlsIW4yywD1XlKtfJc7
           ↪  IJwes/iKrE="
```

```
18      }
19    }
```

Through this function call, the global variable `use_code` henceforth it will be of type `real` (see section 4.8.3).

Server response:

```
1   {
2     "statusCode": 200,
3     "statusMessage": "Success",
4     "type": "SuccessDataResponse"
5   }
```

accept call Request to the server to make the `accept` function call:

```
1   {
2     "message": {
3       "contractInstanceId":
        ↪  "4cc1f3c7-5cb4-4528-b15a-8e5cacf5b18a",
4       "functionName": "accept",
5       "arguments": [
6         {
7           "argument": {
8             "first": "asset",
9             "second": "y",
10            "third": {
11              "ownershipId":
                ↪  "1ce080e5-8c81-48d1-b732-006fa1cc4e2e",
12              "address":
                ↪  "f3hVW1Amltnqe3KvOT00eT7AU23FAUKdgmCluZB+nss=",
13              "unlockScript": "PUSH str
                ↪  CJ3CdFnd6QiRoNaxxJN6sEYkmhKsSKi0SP5YXiSGhygZs+
                ↪  EMyE2bPrI+hRL4PSA0vLh0X6PNpDhTaPxx4kc1LEk9su8+
                ↪  6kkDvi3xpLG9bDoPjss+LEPXUjPTcGVB/3jITb8W+GmX1k
                ↪  DYhGHKtSuhvxBjTwwbtok4gRDD1BcMX/o=\nPUSH str
                ↪  MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDErzzgD2
                ↪  ZslZxciFAiX3/ot7lrkZDw4148jFZrsDZPE6CVs9xXFSHG
                ↪  gy/mFvIFLXhnChO6Nyd2be3lbgeavLMCMVUiTStXr117Km
                ↪  17keWpb3sItkKKsLFBOcIIU8XXowI/OhzQN2XPZYESHgjd
                ↪  Q5vwEj2YyueiS7WKP94YWz/pswIDAQAB\n"
14            }
15          }
16        }
17      ],
18      "type": "FunctionCall"
19    },
20    "signatures": {
```

```
        "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDErzzgD2ZslZxciFAiX
        ↪ 3/ot7lrkZDw4148jFZrsDZPE6CVs9xXFSHGgy/mFvIFLXhnChO6Nyd
        ↪ 2be3lbgeavLMCMVUiTStXr117Km17keWpb3sItkKKsLFBOcIIU8XXo
        ↪ wI/OhzQN2XPZYESHgjdQ5vwEj2YyueiS7WKP94YWz/pswIDAQAB":
        ↪ "wL3r61IgwBGau7S7V967ZSA8B0lLiOMi0qai1YGQVFXnCTvL9WDVM
        ↪ GTwp7XXAQ77f23Hw5y6Ho5SFUMRRfaTLguIJBx9twRSUfpTP4bh3K4
        ↪ RB2yg32rkOP16G2vIfEirTT+v2wmp1f10pY+dY/QdMzua7EFdQNmL7
        ↪ PhJnA96CpM="
    }
}
```

Server response:

```
{
  "statusCode": 200,
  "statusMessage": "Success",
  "type": "SuccessDataResponse"
}
```

Version 1 - end call Request to the server to make the `end` function call:

```
{
  "message": {
    "contractInstanceId":
    ↪ "4cc1f3c7-5cb4-4528-b15a-8e5cacf5b18a",
    "functionName": "end",
    "arguments": [],
    "type": "FunctionCall"
  },
  "signatures": {
    "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDErzzgD2ZslZxciFAiX
    ↪ 3/ot7lrkZDw4148jFZrsDZPE6CVs9xXFSHGgy/mFvIFLXhnChO6Nyd
    ↪ 2be3lbgeavLMCMVUiTStXr117Km17keWpb3sItkKKsLFBOcIIU8XXo
    ↪ wI/OhzQN2XPZYESHgjdQ5vwEj2YyueiS7WKP94YWz/pswIDAQAB":
    ↪ "hU6i0eGRNcZB+ZCxeLCPBM31iai412yczQ4/Td+roq9jnBU7agWfu
    ↪ OyVl/6fCKdTZcKkxASJs1tCpe4bLlpUHt01lFlGM8n9+sPHXl+1/jX
    ↪ MngmmPhuNUPrtsD7PGeFtuC3JJkcqTq3WkyWz6nVdn55bzX6BxleN/
    ↪ I6MPgmDroc="
  }
}
```

Server response:

```
{
  "statusCode": 200,
  "statusMessage": "Success",
  "type": "SuccessDataResponse"
}
```

Version 1 - Trigger of the event and non-execution of the obligation This situation occurs when the customer, who has used the service, has returned the bicycle before the end of use of the service.

From the server logs it can be seen that the event was triggered and the code encoding the obligation was not executed:

```
1  EventTrigger: A new scheduled request has been triggered =>
   ↪ EventTriggerSchedulingRequest{
2    request=CreateEventRequest{
3      obligationFunctionName='obligation_1',
4      time=1680036610
5    },
6    contractId='622ad60b-ab1f-4c2c-9f64-1307c046b55d',
7    contractInstanceId='4cc1f3c7-5cb4-4528-b15a-8e5cacf5b18a'
8  }
9  EventTrigger: Enqueuing the request...
10 EventTrigger: Notifying the virtual machine...
11 EventTrigger: Virtual machine notified
12 EventTrigger: Removing the request from EventTriggerHandler...
13 VirtualMachine: Ready to dequeue a value...
14 VirtualMachine: Request received => Pair{
15   first=null,
16   second=EventTriggerSchedulingRequest{
17     request=CreateEventRequest{
18       obligationFunctionName='obligation_1',
19       time=1680036610
20     },
21     contractId='622ad60b-ab1f-4c2c-9f64-1307c046b55d',
22     contractInstanceId='4cc1f3c7-5cb4-4528-b15a-8e5cacf5b18a'
23   }
24 }
25 VirtualMachine: Just received a trigger request
26 VirtualMachine: This function cannot be called in the current
   ↪ state
27 VirtualMachine: Obligation function name => obligation_1
28 VirtualMachine: Current state => DfaState{name='End'}
29 VirtualMachine: Next state => null
30 VirtualMachine: Ready to dequeue a value...
31 VirtualMachine: I'm waiting...
```

From line 20 to line 22 it can be seen that the obligation has not been performed because the current state of the contract instance is `@End`. Instead, the state in which the obligation should be executed is `@Using`. For this reason it was not possible to fulfill the obligation.

Version 2 - Event trigger and execution of the obligation This situation occurs when the customer, who has used the service, has not returned the bicycle before the end of use of the service.

From the server logs it can be seen that the event was triggered, the code encoding the obligation was loaded and executed:

```
EventTrigger: A new scheduled request has been triggered =>
↪  EventTriggerSchedulingRequest{
  request=CreateEventRequest{
    obligationFunctionName='obligation_1',
    time=1680037664
  },
  contractId='51d909ae-45f8-47d2-90de-40699c8a8a3d',
  contractInstanceId='1a7c6469-b4a3-4c67-8a43-ca60514345f6'
}
EventTrigger: Enqueuing the request...
EventTrigger: Notifying the virtual machine...
EventTrigger: Virtual machine notified
EventTrigger: Removing the request from EventTriggerHandler...
VirtualMachine: Ready to dequeue a value...
VirtualMachine: Request received => Pair{
  first=null,
  second=EventTriggerSchedulingRequest{
    request=CreateEventRequest{
      obligationFunctionName='obligation_1',
      time=1680037664
    },
    contractId='51d909ae-45f8-47d2-90de-40699c8a8a3d',
    contractInstanceId='1a7c6469-b4a3-4c67-8a43-ca60514345f6'
  }
}
VirtualMachine: Just received a trigger request
loadObligationFunction: Loading the obligation function...
loadObligationFunction: Obligation function loaded
VirtualMachine: Function
start:
PUSH real 100 2
GLOAD wallet
GLOAD Lender
WITHDRAW wallet
end:
HALT

loadBytecode: Loading the bytecode...
loadBytecode: Bytecode loaded

VirtualMachine: loadBytecode
start:
PUSH real 100 2
GLOAD wallet
```

```
44    GLOAD Lender
45    WITHDRAW wallet
46    end:
47    HALT
48
49    LegalContractVirtualMachine: execute => Final state of the
      ↪  execution below
50    LegalContractVirtualMachine: execute => The stack is empty
51
52    LegalContractVirtualMachine: execute => GlobalSpace
53    rentingTime: 100, changed: false
54    wallet: 11.00 stipula_coin_asd345, changed: true
55    cost: 12.00, changed: false
56    Borrower: f3hVW1Amltnqe3KvOT00eT7AU23FAUKdgmCluZB+nss=
      ↪  MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDErzzgD2ZslZxciFAiX3
      ↪  /ot7lrkZDw4148jFZrsDZPE6CVs9xXFSHGgy/mFvIFLXhnChO6Nyd2be3l
      ↪  bgeavLMCMVUiTStXr117Km17keWpb3sItkKKsLFBOcIIU8XXowI/OhzQN2
      ↪  XPZYESHgjdQ5vwEj2YyueiS7WKP94YWz/pswIDAQAB, changed:
      ↪  false
57    use_code: 1.00, changed: false
58    Lender: ubL35Am7TimL5R4oMwm2OxgAYA3XT3BeeDE56oxqdLc=
      ↪  MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCo/GjVKS+3gAA55+kko4
      ↪  1yINdOcCLQMSBQyuTTkKHE1mhu/TgOpivM0wLPsSga8hQMr3+v3aR0IF/v
      ↪  fCRf6SdiXmWx/jflmEXtnT6fkGcnV6dGNUpHWXSpwUIDt0N88jfnEqekx4
      ↪  S+KDCKg99sGEeHeT65fKS8lB0gjHMt9AOriwIDAQAB, changed:
      ↪  false
59
60    LegalContractVirtualMachine: execute => The argument space is
      ↪  empty
61
62    LegalContractVirtualMachine: execute => The data space is empty
63
64    Global state of the execution
65    running -> false
66    executionPointer -> 6
67    executionPointer (with offset) -> 67
68    length of the program -> 7
69    length of the program (with offset) -> 68
70    VirtualMachine: Updating the global store...
71    VirtualMachine: Global store updated
72    VirtualMachine: Ready to dequeue a value...
73    VirtualMachine: I'm waiting...
```

In this case, however, it was possible to execute the code that encodes the obligation because the current state of the contract instance is `@Using` and coincides with the state in which the obligation is to be performed.

Version 2 - end call Here we show the example in which the user tries to call the `end` function, to notify the company of the end of using the service. However, the call to this function took place after the maximum time established by the contract, and therefore the penalty foreseen by the contract was activated.

Request to the server to make the `end` function call:


```
{
  "message": {
    "contractInstanceId":
    ↪  "1a7c6469-b4a3-4c67-8a43-ca60514345f6",
    "functionName": "end",
    "arguments": [],
    "type": "FunctionCall"
  },
  "signatures": {
    "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDErzzgD2ZslZxciFAiX
    ↪  3/ot7lrkZDw4148jFZrsDZPE6CVs9xXFSHGgy/mFvIFLXhnChO6Nyd
    ↪  2be3lbgeavLMCMVUiTStXr117Km17keWpb3sItkKKsLFBOcIIU8XXo
    ↪  wI/OhzQN2XPZYESHgjdQ5vwEj2YyueiS7WKP94YWz/pswIDAQAB":
    ↪  "Ow8gS8d5SChD3E5CgtsnFHTRWskdeWW2IsTLQJk92mS40LfVtPcxu
    ↪  DiIbzL7xWwtUTMFxza+/TSxU+rMsVvMqQLLyUQ4e6UrLO25+Nr7p5x
    ↪  013JGIaxc18G5kqEuS4iEyiqN1479E4ElLROE+VpI5DBAKMegw0h9m
    ↪  5cbtHFN/fA="
  }
}
```


Server response:


```
{
  "data": "This function cannot be called in the current
  ↪  state",
  "statusCode": 404,
  "statusMessage": "Success",
  "type": "SuccessDataResponse"
}
```


Single-use-seals by Lender Server request to get `Lender`'s single-use-seals:


```
{
  "message": {
    "address": "ubL35Am7TimL5R4oMwm2OxgAYA3XT3BeeDE56oxqdLc=",
    "type": "GetOwnershipsByAddress"
  },
  "signatures": {
    "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCo/GjVKS+3gAA55+kko
    ↪  41yINdOcCLQMSBQyuTTkKHE1mhu/TgOpivM0wLPsSga8hQMr3+v3aR
    ↪  0IF/vfCRf6SdiXmWx/jflmEXtnT6fkGcnV6dGNUpHWXSpwUIDt0N88
    ↪  jfnEqekx4S+KDCKg99sGEeHeT65fKS8lB0gjHMt9AOriwIDAQAB":
    ↪  "MomZTc63z7PfH35c1dL4tjXebcsW+0Zxl0nP1NQdcUFws98DX+bMW
    ↪  I7L0C6IO5lxvkYve4zdio1Crn97FXvngK4aVfiEZEnHOJ0tstq7uQY
    ↪  GErM3DDAABqPq8HH5yoKnLST2LWpO0oD8G/VXvIE6qMT5D34W1Ci0q
    ↪  4uh+7y3EcY="
```

```
    }
  }
```

Server response:

```
{
  "data": "[
      Ownership{
        id='205cd89a-c078-4f1a-8dd7-dae683c4f3a8',
        singleUseSeal=SingleUseSeal{
          assetId='stipula_coin_asd345',
          amount=RealType{
            value=100,
            decimals=2
          },
          lockScript='DUP\nSHA256\nPUSH str
          ↪  ubL35Am7TimL5R4oMwm2OxgAYA3XT3BeeDE56oxqdLc=\nEQ
          ↪  UAL\nCHECKSIG\nHALT\n'
        },
        unlockScript='',
        contractInstanceId=''
      }
    ]",
  "statusCode": 200,
  "statusMessage": "Success",
  "type": "SuccessDataResponse"
}
```

It is possible to notice that the `Lender` has received a new single-use-seal from the agreement instance.

Single-use-seals by Borrower Server request to get `Borrower`'s single-use-seals:

```
{
  "message": {
    "address": "f3hVW1Amltnqe3KvOT00eT7AU23FAUKdgmCluZB+nss=",
    "type": "GetOwnershipsByAddress"
  },
  "signatures": {
    "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDErzzgD2ZslZxciFAiX
    ↪  3/ot7lrkZDw4148jFZrsDZPE6CVs9xXFSHGgy/mFvIFLXhnChO6Nyd
    ↪  2be3lbgeavLMCMVUiTStXr117Km17keWpb3sItkKKsLFBOcIIU8XXo
    ↪  wI/OhzQN2XPZYESHgjdQ5vwEj2YyueiS7WKP94YWz/pswIDAQAB":
    ↪  "hSNodnUyusffNlv+KNq4605pFvqh91pVspFhTgbmWccE/LKM6h4be
    ↪  dpvTgMHoVDezvA7v2XTzmLG5eL3lOeA6I2xJMH32DcV60IPSoh61oV
    ↪  HnwPQcQHY039D4y5VSJ0GMQJKIcTEq3fqIdabg7261xUaegHUnXrcy
    ↪  ynh9GpMJxk="
  }
}
```

Server response:

```
{
  "data": "[
      Ownership{
        id='1ce080e5-8c81-48d1-b732-006fa1cc4e2e',
        singleUseSeal=SingleUseSeal{
          assetId='stipula_coin_asd345',
          amount=RealType{
            value=1200,
            decimals=2
          },
          lockScript='DUP\nSHA256\nPUSH str
          ↪  f3hVW1Amltnqe3KvOT00eT7AU23FAUKdgmCluZB+nss=\nEQ
          ↪  UAL\nCHECKSIG\nHALT\n'
        },
        unlockScript='PUSH str CJ3CdFnd6QiRoNaxxJN6sEYkmhKsSKi
        ↪  0SP5YXiSGhygZs+EMyE2bPrI+hRL4PSA0vLh0X6PNpDhTaPxx4
        ↪  kc1LEk9su8+6kkDvi3xpLG9bDoPjss+LEPXUjPTcGVB/3jITb8
        ↪  W+GmX1kDYhGHKtSuhvxBjTwwbtok4gRDD1BcMX/o=\nPUSH
        ↪  str MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDErzzgD2
        ↪  ZslZxciFAiX3/ot7lrkZDw4148jFZrsDZPE6CVs9xXFSHGgy/m
        ↪  FvIFLXhnChO6Nyd2be3lbgeavLMCMVUiTStXr117Km17keWpb3
        ↪  sItkKKsLFBOcIIU8XXowI/OhzQN2XPZYESHgjdQ5vwEj2Yyuei
        ↪  S7WKP94YWz/pswIDAQAB\n',
        contractInstanceId='4cc1f3c7-5cb4-4528-b15a-8e5cacf5b1
        ↪  8a'
      }
    ]",
  "statusCode": 200,
  "statusMessage": "Error",
  "type": "ErrorDataResponse"
}
```

It can be seen that the single-use-seal has been spent and is demonstrated by the `unlockScript` field.

Appendix B

Grammar

B.1 Lexer rules

```
1   SEMIC    : ';' ;
2   COLON    : ':' ;
3   COMMA    : ',' ;
4   DOT      : '.' ;
5   EQ       : '==' ;
6   NEQ     : '!=' ;
7   IMPL     : '==>' ;
8   ASM      : '=' ;
9   ASSETUP :  '-o' ;
10  FIELDUP : '->' ;
11  PLUS     : '+' ;
12  MINUS    : '-' ;
13  TIMES    : '*' ;
14  DIV      : '/' ;
15  AT       : '@' ;
16  TRUE     : 'true' ;
17  FALSE    : 'false' ;
18  LPAR     : '(' ;
19  RPAR     : ')' ;
20  SLPAR   : '[' ;
21  SRPAR   : ']' ;
22  CLPAR    : '{' ;
23  CRPAR    : '}' ;
24  LEQ    : '<=';
25  GEQ    : '>=';
26  LE    : '<';
27  GE    : '>';
28  OR    : '||';
29  AND    : '&&';
30  NOT    : '!';
31  EMPTY   : '_' ;
32  NOW   : 'now' ;
33  TRIGGER : '>>';
```

```
34  IF     : 'if' ;
35  ELSEIF   : 'else if' ;
36  ELSE   : 'else' ;
37  STIPULA : 'stipula';
38  ASSET : 'asset' ;
39  FIELD : 'field' ;
40  AGREEMENT : 'agreement';
41  INTEGER : 'int' ;
42  DOUBLE : 'real' ;
43  BOOLEAN : 'bool' ;
44  STRING : 'string' ;
45  PARTY : 'party' ;
46  INIT : 'init' ;
47
48  RAWSTRING : '\'' ~('\'')+ '\'' | '"' ~('"')+ '"' ;
49
50  INT : '0' | [1-9] [0-9]* ;
51
52  REAL : [0-9]* '.' [0-9]+ ;
53
54  WS
55  : [ \t\r\n] -> skip
56  ;
57
58  //IDs
59  fragment CHAR  : 'a'..'z' |'A'..'Z' ;
60  ID : CHAR (CHAR | INT | EMPTY)* ;
61
62  OTHER
63  : .
64  ;
65
66  //ESCAPED SEQUENCES
67  LINECOMENTS    : '//' (~('\n'|'\r'))* -> skip;
68  BLOCKCOMENTS    : '/*'(
    ↪  ~('/'|'*')|'/'~'*'|'*'~'/'|BLOCKCOMENTS)* '*/' -> skip;
```

B.2 Parser rules

```
1   prog :  STIPULA contract_id = ID CLPAR (assetdecl)?
    ↪  (fielddecl)? INIT init_state = ID agreement (fun)+ CRPAR ;
2
3   agreement : (AGREEMENT LPAR party (COMMA party)* RPAR LPAR
    ↪  vardec (COMMA vardec)* RPAR CLPAR (assign)+ CRPAR IMPL AT
    ↪  state);
4
5   assetdecl :  ASSET idAsset+=ID ':' assetId+=ID (','
    ↪  idAsset+=ID ':' assetId+=ID)* ;
6
7   fielddecl : FIELD idField+=ID (',' idField+=ID)* ;
```

```
8
9   fun  :  ((AT state)* party (COMMA party)* COLON funId=ID LPAR
    ↪  (vardec ( COMMA vardec)* )? RPAR SLPAR (assetdec ( COMMA
    ↪  assetdec)* )? SRPAR (LPAR prec RPAR)? CLPAR (stat)+ SEMIC
    ↪  (events)+ CRPAR IMPL AT state )   ;
10
11  assign : (party (COMMA party)* COLON vardec (COMMA vardec)*);
12
13  dec : (ASSET | FIELD) ID  ;
14
15  type :  INTEGER | DOUBLE | BOOLEAN | STRING ;
16
17  state : ID;
18
19  party : ID;
20
21  vardec  : ID ;
22
23  assetdec  : ID ;
24
25  varasm     : vardec ASM expr ;
26
27  stat   :      EMPTY
28        |   left=value operator=ASSETUP right=ID (COMMA
          ↪  rightPlus=ID)?
29        |   left=value operator=FIELDUP right=(ID | EMPTY)
30        | ifelse
31
32        ;
33
34  ifelse : (IF LPAR cond=expr RPAR CLPAR ifBranch+=stat
    ↪  (ifBranch+=stat)* CRPAR (ELSEIF condElseIf+=expr CLPAR
    ↪  elseIfBranch+=stat (elseIfBranch+=stat)* CRPAR)* (ELSE
    ↪  CLPAR elseBranch+=stat (elseBranch+=stat)* CRPAR )?);
35
36  events  :     EMPTY
37        | ( expr TRIGGER AT ID CLPAR stat+ CRPAR IMPL AT ID )
38        ;
39
40  prec  : expr
41      ;
42
43  expr   :  ('-')? left=term (operator=(PLUS | MINUS | OR)
    ↪  right=expr)?
44        ;
45
46  term   : left=factor (operator=(TIMES | DIV  | AND)
    ↪  right=term)?
47        ;
48
```

```
49  factor : left=value (operator = (EQ | LE | GE | LEQ | GEQ |
    ↪  NEQ ) right=value)?
50        ;
51
52  value  :  number
53      | ID
54      | NOW
55        | LPAR expr RPAR
56        | RAWSTRING
57        | EMPTY
58        | (TRUE | FALSE)
59        ;
60
61  real : number DOT number ;
62
63  number : INT | REAL ;
```

Appendix C

Pipelines

C.1 run-tests.yml

```
name: Tests

on:
  push:
    branches: [ master, test ]
  pull_request:
    types:
      - closed
    branches:
      - master

jobs:
  run-tests:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v3
      - name: Set up JDK 8
        uses: actions/setup-java@v3
        with:
          java-version: '8'
          distribution: 'corretto'
      - name: Validate Gradle wrapper
        uses: gradle/wrapper-validation-action@e6e38bacfdf1a33⌋
        ↪ 7459f332974bb2327a31aaf4b
      - name: Build with Gradle
        uses: gradle/gradle-build-action@67421db6bd0bf253fb4bd⌋
        ↪ 25b31ebb98943c375e1
        with:
          arguments: build
          version: 7.6.0-jdk8
```

C.2 create-and-push-docker-image.yml

```
name: Create and publish a Docker image

on:
  push:
    tags: [ v* ]

env:
  REGISTRY: ghcr.io
  IMAGE_NAME: ${{ github.repository }}

jobs:
  build-and-push-image:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      packages: write

    steps:
      - name: Checkout repository
        uses: actions/checkout@v3

      - name: Log in to the Container registry
        uses: docker/login-action@f054a8b539a109f9f41c372932f1ae047eff08c9
        with:
          registry: ${{ env.REGISTRY }}
          username: ${{ github.actor }}
          password: ${{ secrets.GITHUB_TOKEN }}

      - name: Extract metadata (tags, labels) for Docker
        id: meta
        uses: docker/metadata-action@98669ae865ea3cffbcbaa878cf57c20bbf1c6c38
        with:
          images: ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}

      - name: Build and push Docker image
        uses: docker/build-push-action@ad44023a93711e3deb338508980b4b5e9bcdc5dc
        with:
          context: .
          push: true
          tags: ${{ steps.meta.outputs.tags }}
          labels: ${{ steps.meta.outputs.labels }}
```

Appendix D

Gradle

D.1 build.gradle

```
1  plugins {
2    id 'antlr'
3    id 'java'
4  }
5
6  group 'org.example'
7  version '1.0-SNAPSHOT'
8
9  repositories {
10     mavenCentral()
11 }
12
13 dependencies {
14     implementation 'com.google.code.gson:gson:2.10.1'
15     implementation 'org.iq80.leveldb:leveldb-api:0.9'
16     implementation 'org.iq80.leveldb:leveldb:0.9'
17
18     implementation 'org.antlr:antlr4-runtime:4.10'
19     antlr 'org.antlr:antlr4:4.10'
20
21     testImplementation
       ↪  'org.junit.jupiter:junit-jupiter-api:5.8.1'
22     testRuntimeOnly
       ↪  'org.junit.jupiter:junit-jupiter-engine:5.8.1'
23 }
24
25 test {
26     useJUnitPlatform()
27 }
28
29 jar {
30     manifest {
31         attributes "Main-Class": "Main"
```

```
32          }
33
34          from {
35              configurations.runtimeClasspath.collect {
                ↪  it.isDirectory() ? it : zipTree(it) }
36          }
37      }
38
39      generateGrammarSource {
40          maxHeapSize = "64m"
41          arguments += ["-visitor"]
42      }
```

Appendix E

Docker

E.1 Dockerfile

```
# Setup Gradle
FROM gradle:7.6.0-jdk8 AS TEMP_BUILD_IMAGE
ENV APP_HOME=/usr/app
WORKDIR $APP_HOME
COPY build.gradle settings.gradle $APP_HOME

COPY gradle $APP_HOME/gradle
COPY --chown=gradle:gradle . /home/gradle/src
USER root
RUN chown -R gradle /home/gradle/src

RUN gradle build || return 0
COPY . .
RUN gradle clean build

# Setup Java
FROM amazoncorretto:8
ENV ARTIFACT_NAME=stipula-node-1.0-SNAPSHOT.jar
ENV APP_HOME=/usr/app

WORKDIR $APP_HOME
COPY --from=TEMP_BUILD_IMAGE
↪  $APP_HOME/build/libs/$ARTIFACT_NAME .

# Run
EXPOSE 8080
ENTRYPOINT exec java -jar ${ARTIFACT_NAME}
```

E.2 docker-compose.yml

```
1  version: "3.3"
2  services:
3    node:
4      container_name: "stipula-node"
5      image: stipula-node:v0.4.2
6      ports:
7        - 127.0.0.1:8080:8080
8        - 127.0.0.1:61000:61000
9      volumes:
10       - stipula-storage:/usr/app/storage/
11     environment:
12       - SEED=no
13
14 volumes:
15   stipula-storage:
```

If you don't want to manually build the Docker image, you can replace line 5 with `image: "ghcr.io/federicozanardo/stipula-node:v0.4.2"`, this will download a specific image from a specific GitHub page (Zanardo, 2023a).

Bibliography

References

A. Das, S. Balzer, J. Hoffmann, F. Pfenning and I. Santurkar (2021). Resource-Aware Session Types for Digital Contracts. *2021 2021 IEEE 34th Computer Security Foundations Symposium (CSF), IEEE Computer Society*, 111–126. URL: https://doi.org/10.1109/CSF51468.2021.00004.

Antonopoulos, Andreas M. (2017). *Mastering Bitcoin. Programming the open blockchain.* Second Edition. O'Reilly. Chap. 6,10, pp. 131–138, 215–237.

David Siegel (2016). Understanding the dao attack. *First Monday 21(12)*. URL: doi:10.5210/fm.v21i12.7113.

Franklin Schrans, Susan Eisenbach and Sophia Drossopoulou (2018). Writing Safe Smart Contracts in Flint. *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming, Programming'18 Companion, ACM*, 218–219. URL: https://doi.org/10.1145/3191697.3213790.

Lawrence Lessig (1999). Code and Other Laws of Cyberspace. *Basic Books, Inc.*

Primavera De Filippi and Samer Hassan (2016). Blockchain technology as a regulatory technology: From code is law to law is code. URL: https://top-forex-brokers.com/%202021/10/07/understanding-the-dao-attack/.

Sam Blackshear and et al. (2021). Move: A Language With Programmable Resources. URL: https://developers.diem.com/papers/diem-move-a-language-with-programmable-resources/2020-04-09.pdf.

Shrutarshi Basu, Anshuman Mohan, James Grimmelmann and Nate Foster (2022). Legal Calculi. *ProLaLa Programming Languages and the Law*. URL: https://popl22.sigplan.org/details/prolala-2022-papers/6/Legal-Calculi.

Silvia Crafa (2022). From Legal Contracts to Legal Calculi: the code-driven normativity. *Proc. of Workshop EXPRESS/SOS 2022* **368 EPTCS**, 23–42. URL: doi.org/10.4204/EPTCS.368.2.

Silvia Crafa, Cosimo Laneve and Giovanni Sartor (2021). Pacta sunt servanda: legal contracts in Stipula. *arXiv*. URL: https://arxiv.org/abs/2110.11069.

— (2022). Stipula: a domain specific language for legal contracts. *Technical Report, ProLaLa 2022 ProLaLa Programming Languages and the Law*. URL: https://popl22.sigplan.org/details/prolala-2022-papers/6/Legal-Calculi.

Vimal Dwivedi, Vishwajeet Pattanaik, Vipin Deval, Abhishek Dixit, Alex Norta and Dirk Draheim (2021). Legally Enforceable Smart-Contract Languages: A Systematic Literature Review. *ACM Comput.* URL: https://doi.org/10.1145/3453475.

Sitography

References

Adam Back (2002). *Hashcash - a denial of service counter-measure*. URL: https://www.hashcash.org/papers/hashcash.pdf.

Akoma Ntoso (2018). Akoma Ntoso XML for parliamentary, legislative and judiciary documents. URL: http://www.%20akomantoso.org/.

ANTLR v4.10. ANTLR official site. URL: https://github.com/antlr/website-antlr4/blob/gh-pages/download/antlr-4.10-complete.jar.

ANTLR. ANother Tool for Language Recognition. ANTLR official site. URL: https://www.antlr.org/.

Arbitrum. Arbitrum official site. URL: https://arbitrum.io/.

Arbitrum fees. Arbitrum official site. URL: https://developer.arbitrum.io/arbos/gas.

Bitcoin halving explained. Coindesk official site. URL: https://www.coindesk.com/learn/bitcoin-halving-explained/.

Bitcoin mining and halving. Cointelegraph official site. URL: https://cointelegraph.com/learn/bitcoin-halving-how-does-the-halving-cycle-work-and-why-does-it-matter.

Catala in action (2022). Language site. URL: https://catala-lang.org/.

Cracking the Code (2020). Cracking the Code: Rulemaking for humans and machines. URL: https://oecd-opsi.org/publications/cracking-the-code/.

Docker: Accelerated, Containerized Application Development. Docker official site. URL: https://www.docker.com/.

GitHub. GitHub official site. URL: https://github.com/.

IntelliJ IDEA. IntelliJ IDEA official site. URL: https://www.jetbrains.com/idea/.

Java. Java official site. URL: https://www.java.com/.

LevelDB Official Repository. Official GitHub repository. URL: https://github.com/google/leveldb.

Lightning Network. URL: https://lightning.network/.

Nick Szabo (1997). *Formalizing and Securing Relationships on Public Networks*. URL: https://bitcoinstan.io/prehistory/doc/1997_2.pdf.

Obsidian: A safer blockchain programming language (2018). Official Obsidian site. URL: http://obsidian-lang.com/.

Optimism. Optimism official site. URL: https://www.optimism.io/.

Optimism fees. Optimism official site. URL: https://help.optimism.io/hc/en-us/articles/4411895794715-How-do-transaction-fees-on-Optimism-work-.

Plasma chains. URL: https://ethereum.org/en/developers/docs/scaling/plasma/.

Polygon Technology. URL: https://polygon.technology/.

Silvia Crafa and Cosimo Laneve (2022). *Programming legal contracts: A beginners guide to Stipula*. URL: https://cs.unibo.it/~laneve/papers/beginStipula.pdf.

Silvia Crafa, Cosimo Laneve and Adele Veschetti (2022a). *Stipula Prototype*. GitHub Repository. URL: https://github.com/stipula-language.

— (2022b). *Stipula Prototype*. GitHub Repository. URL: https://github.com/stipula-language/stipula/blob/master/Stipula-LAN/Stipula.g4.

— (2022c). *Stipula Prototype*. GitHub Repository. URL: https://github.com/stipula-language/stipula/blob/6e98b56bb10403eb5e23ec8a8ef832dee1ff51d8/syntax.pdf.

Solidity Documentation: State Machine Common Pattern. Official Solidity site. URL: https://docs.soliditylang.org/en/v0.8.0/common-patterns.html#state-machine.

The CoHuBiCoL research project (2019). URL: https://www.cohubicol.com/about.

The European ODR platform. Official site. URL: https://ec.europa.eu/consumers/odr.

Zanardo, Federico (2023a). *Stipula available packages*. Official GitHub repository. URL: https://github.com/federicozanardo/stipula-node/pkgs/container/stipula-node.

— (2023b). *Stipula graph dependencies*. Official GitHub repository. URL: https://github.com/federicozanardo/stipula-node/tree/master/graphs.

— (2023c). *Stipula implementation*. Official GitHub repository. URL: https://github.com/federicozanardo/stipula-node.

— (2023d). *Stipula issues section*. Official GitHub repository. URL: https://github.com/federicozanardo/stipula-node/issues.

— (2023e). *Stipula last release*. Official GitHub repository. URL: https://github.com/federicozanardo/stipula-node/releases/tag/v0.4.2.

— (2023f). *Stipula milestones section*. Official GitHub repository. URL: https://github.com/federicozanardo/stipula-node/milestones.

— (2023g). *Stipula packages page*. Official GitHub repository. URL: https://github.com/federicozanardo?tab=packages&repo_name=stipula-node.

— (2023h). *Stipula releases*. Official GitHub repository. URL: https://github.com/federicozanardo/stipula-node/releases.