WebAssembly Execution Environment for Dela

Maxime Sierro

School of Computer and Communication Sciences

Decentralized and Distributed Systems lab

Master Semester Project

June 2021

Responsible
Prof. Bryan Ford
EPFL / DEDIS

Supervisor
Noémien Kocher
EPFL / DEDIS

Contents

1 Introduction

The goal of the project is to implement a smart contract execution environment which uses WebAssembly [1] to handle smart contracts written in different languages. It must communicate with the DELA framework [2], which is a blockchain-based distributed ledger currently developed in Go by the Decentralized and Distributed Systems lab. Only a small subset of said framework must receive changes to communicate with the new environment. As a result, the vast majority of the work is focused on the environment itself, which is implemented from scratch.

1.1 Motivation

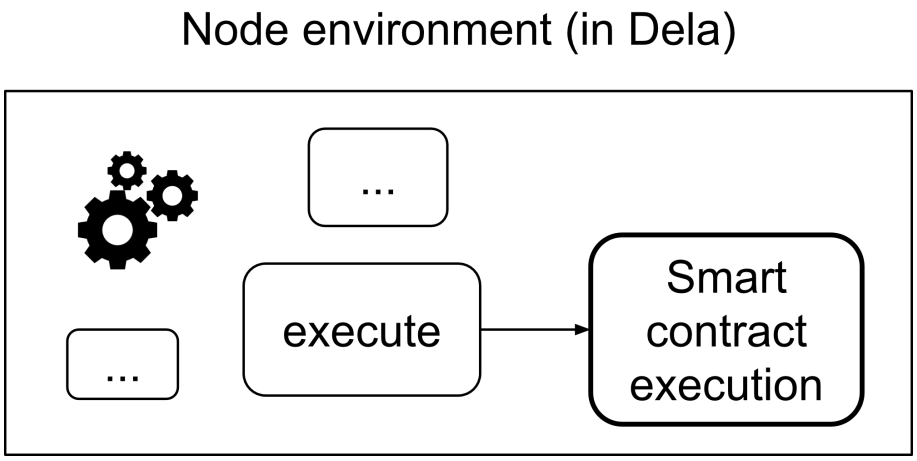

Figure 1: Native smart contracts execution environment.

As shown in figure 1, the standard way of executing a smart contract in DELA is inside of a node's environment, which we call "native" execution. While this method is very intuitive and efficient, it has two main limitations. The first one is that adding, removing or modifying a smart contract requires re-compiling the entirety of the environment, which is highly unpractical in a realistic blockchain scenario where such changes are frequent. The second is that smart contracts are required to be pre-compiled for the environment running the ledger and must thus be written in the same language, which is Go in this case.

To circumvent the first limitation, the obvious solution is to decouple the smart contract execution environment from the node's environment, as illustrated in figure 2. Any modification on the smart contracts would only require recompiling the smaller, decoupled environment which opens up the possibility of loading and unloading smart contracts dynamically without interrupting the main DELA environment. With this decoupled environment,

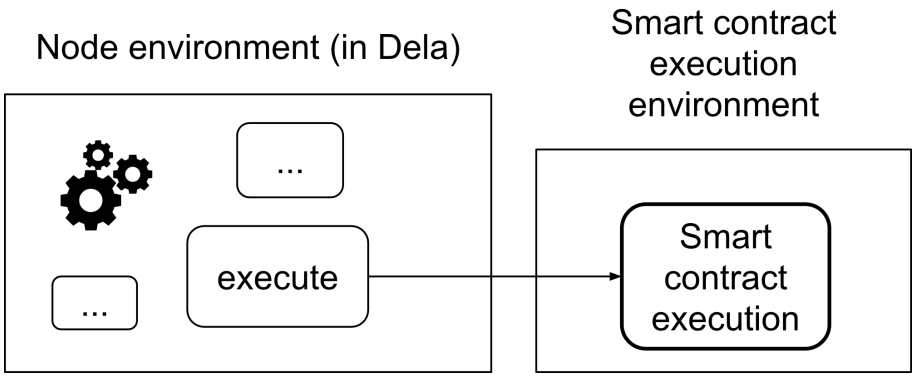

Figure 2: Decoupled smart contracts execution environment.

smart contracts must not necessarily be written in Go, which somewhat takes care of the second limitation. However, supporting multiple languages efficiently is not straightforward and requires something else, which is when WebAssembly comes into the picture.

WebAssembly (WASM) is an open standard that defines a binary format which can obtained from higher level compatible languages. As its name implies, it is made up of "Assembly-like" instructions which can thus be run on a wide range of machines very efficiently, thanks to their low level. The WASM code is executed in a sandboxed environment. This added flexibility, its host independence and low performance penalty make it an ideal candidate for a smart-contract runtime used by many nodes trying to reach a consensus. Ethereum's proposed redesign of their execution layer using WASM further indicates that there is great potential for the technology in the blockchain field.

1.2 Goals

1. Implementation of a fully functional execution module that uses WASM
2. Simultaneous support of multiple smart contracts written in different languages
3. Justification of configuration choices and the selection of compatible languages
4. Performance comparisons
5. Determinism analysis
6. Testing comparable to the native module

2 Design

2.1 General Configuration

The most important design decision during the project was deciding what "kind" of environment should be built. WebAssembly was introduced in 2017, when its initial aim was to enable high performance web-browser applications. Over time however, WASM started seeing use outside of browsers because of its many advantages, especially its near-native performance and consistent execution across different hardwares. As a result, three options were identified and considered : a web browser application, a web server and a unix daemon.

The first factor that was taken into account was the general lack of resources on WASM, which is severely exacerbated when the environment is not browser-based. Even though a web-browser application functionally made less sense than the other options since there is no human interaction, it was the chosen solution at the start of the project. The daemon would have communicated with the DELA framework using unix sockets and would have used a runtime like Wasmer [3], Wasmtime [4] or WAVM [5].

8 Bibliography

References

[1] Webassembly. `https://webassembly.org`.

[2] Dedis ledger architecture. `https://github.com/dedis/dela`.

[3] Wasmer - the universal webassembly runtime. `https://wasmer.io/`.

[4] "wasmtime - a small and efficient runtime for webassembly and wasi". https://wasmtime.dev/.

[5] Wavm - webassembly virtual machine. https://wavm.github.io/.