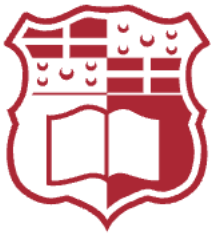

L-Università ta' Malta

DLT5400 - DLT Implementation and Internals

Assignment 2 - (Substrate JVM Interpreter)

Balaganapathy Vanagiri Selvakumar

Centre for Distributed Ledger Technologies

June 13, 2023

1 Q1 - Smart Contract Upload and Retrieval

1.1 Pre - Requisites

In order to successfully run a substrate node, we need to have some pre installed tools and softwares. Run the below commands in order to install rust and set up the environment to run a substrate node

```
brew update
brew install openssl
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
source ~/.cargo/env
rustc --version
rustup default stable
rustup update
rustup target add wasm32-unknown-unknown
rustup show
rustup +nightly show
brew install cmake
```

After the above installations, we can now install and run a substrate node. Let's clone the substrate repository in our working directory.

1.2 Substrate node with Front-end

Lets now try running a substrate node instance with an integrated front end.

```
git clone https://github.com/substrate-developer-hub/substrate-node-template
cd substrate-node-template
cargo build --release
```

The last command will build the substrate node with the template pallet integrated. Execute the below command to start a node in a new terminal window from the project directory.

```
./target/release/node-template --dev
```

Now to integrate the substrate instance with a front-end, let's use the node template provided by the substrate.

```
git clone https://github.com/substrate-developer-hub/substrate-front-end-template.git
cd substrate-front-end-template
yarn install
yarn start
```

The above command clones the template repository installs the yarn dependencies and starts the yarn front-end instance.

1.3 Custom Smart Contract Storage Pallet

We are going to modify the template pallet that is provided with the repository and use it as a custom pallet to store and retrieve smart contracts in byte code format. Open the pallets folder from your project directory and go inside the template folder. You can find a file named *lib.rs* inside the *src* folder. Let us modify the code present in this file step by step.

```
use sp_std::vec::Vec;
use const_random::const_random;
```

Firstly, let us add some necessary dependencies that will be used in the following code implementations. We add the Vector datatype and a random generator dependency.

```
#[pallet::storage]
pub type SC<T> = StorageMap<_,Twox64Concat, Vec<u8>, String>;
```

Now we add a storage variable for the pallet to hold a map of the smart contract address and its code respectively. We create *StorageMap*, which is a functionality provided by the substrate to store key-value pairs. We declare a map with a *Vec<u8>* type key which is mapped to a *String* value that stores the byte code representation of the smart contract.

```
SCUpload {address: Vec<u8>, who: T::AccountId},
SCRetrived {contract: String},
    SCExecuted {output: String, bno: Option<u32>},
```

We create three events that can be emitted to the blockchain to report or log the function activities. We emit an event when a smart contract is uploaded with its address, an event when the smart contract is retrieved with its smart contract byte code value and an event when we successfully execute the contract uploaded.

```
#[pallet::call_index(0)]
#[pallet::weight(T::WeightInfo::do_something())]
pub fn upload(origin: OriginFor<T>, code: String>) -> DispatchResult {
  let who = ensure_signed(origin)?;
  let mut hasher = Sha256::new();
          hasher.update(&bytecode);
          let address: Vec<u8> = hasher.finalize().to_vec();
  SC::<T>::insert(&address,code);
  Self::deposit_event(Event::SCUpload { address, who });
  Ok(())
```

```
}
```

Now, we create a function to upload our smart contracts and store them in the substrate blockchain. We have used a *SHA256* hashing to generate a unique smart contract address. We insert this address as the key and the byte code representation of the smart contract passed as an argument to the function as the value of our SmartContracts map. Finally, we emit the *SCUpload* event that logs the smart contract address onto the blockchain and the front end for our later usage.

```
#[pallet::call_index(1)]
#[pallet::weight(T::WeightInfo::do_something())]
pub fn retrive(origin: OriginFor<T>, arg: Vec<u8>) -> DispatchResult {
  let _who = ensure_signed(origin)?;
  let contract = SC::<T>::get(arg).ok_or(Error::<T>::NoneValue)?;
  Self::deposit_event(Event::SCRetrived { contract });
  Ok(())
}
```

Then, we add a function to handle the retrieval of the smart contracts stored. We pass the address generated in the *upload* function as an argument and get the value associated with it from the map. We throw an error if the smart contract doesn't exist. We emit an event with the smart contract byte code to the blockchain that can be viewed from the front-end instance.

Finally, we have our execute function which executes the bytecode stored in a particular address and outputs the result. The implementation of this function will be discussed in detail in the below section 2

1.4 Using the custom pallet

To test our custom pallet and use it, we need to build and run our substrate node again. Before that let us add a few dependencies in the *cargo.toml* file inside the template folder.

```
sp-std = { version = "5.0.0", default-features = false, git =
    "https://github.com/paritytech/substrate.git", branch = "polkadot-v0.9.42" }
const-random = "0.1.15"
```

This will download the respective crates and dependencies required for our pallet to function properly. Now let's build and run the substrate node.

```
cargo build --release
./target/release/node-template --dev
```

You can now see our node is up and the blocks getting created in the terminal and our front-end instance.

Figure 1: Substrate node started

Now let us try out our pallet from the front-end instance. Scroll down to the pallet interactor panel in our front end. Select *TemplateModule* in the available pallets dropdown. Now choose the upload function in the callables and enter your smart contract's byte code. We have entered the custom bytecode that will be discussed in the upcoming section. Click the signed button to send a signed transaction to the blockchain. Now you can see the event *SCUpload* in the right side events panel.

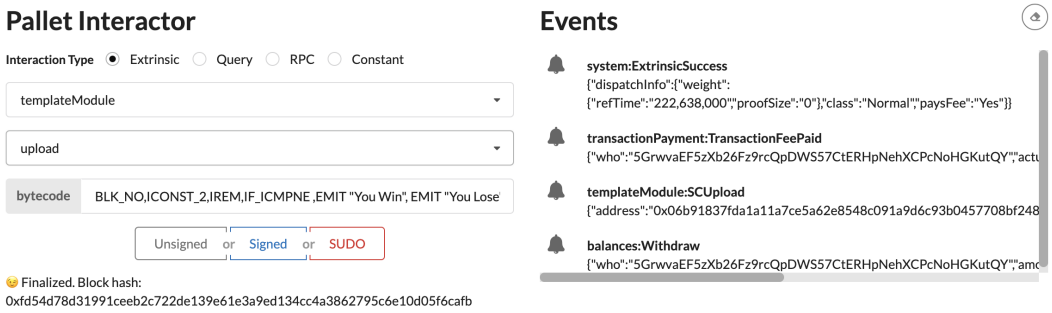

Figure 2: Upload function call

To fetch the smart contract from the blockchain, copy the address from the Events pane and choose the retrieve function in the callables. Now paste the address in the argument text box and hit the signed button. This will return the smart contract bytecode that was stored in the events pane.

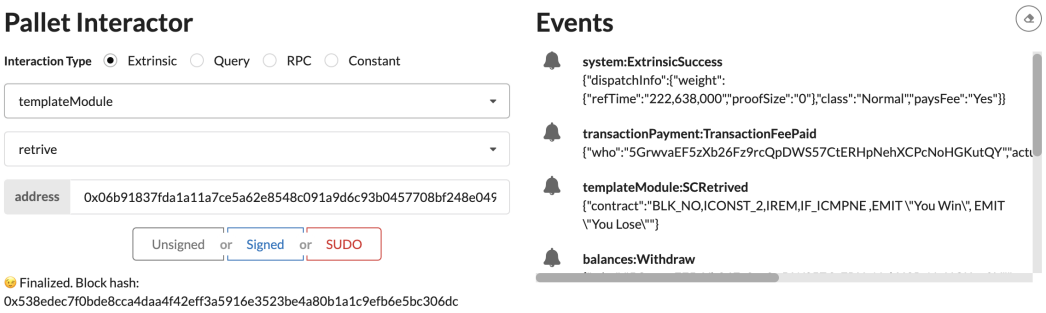

Figure 3: Retrieve function call

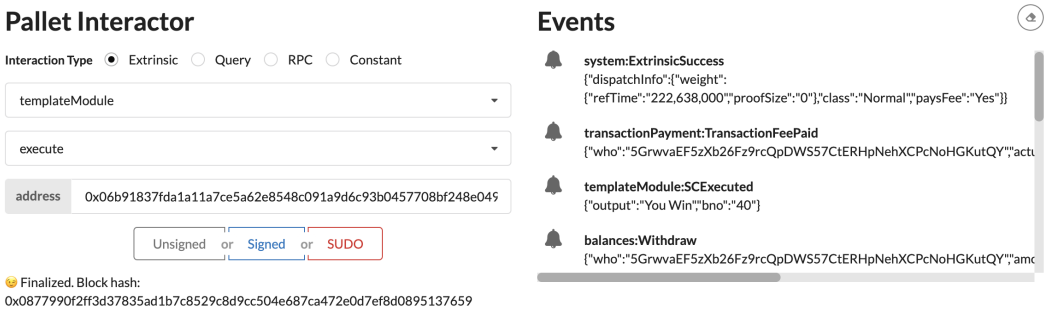

Figure 4: Execute function call

2 Q2 - JVM Intrepreter

2.1 Introduction

Our custom substrate pallet now has a function to handle smart contract upload and retrieval. We store our smart contracts in a custom JVM Style bytecode which will be discussed in this section. We will also discuss the implementation of a very limited interpreter which executes the smart contract in a particular address and emits the output as an event to the blockchain.

2.2 Bytecode Discussion

As per the specification document, we have implemented some custom bytecode specifications to facilitate some specific operations.

```
SOURCE CODE:
if (blockNumber % 2 == 0) {
    emit "You win";
} else {
    emit "You lose";
}


BYTECODE IMPLEMENTATION:
BLK_NO,ICONST_2,IREM,IF_ICMPNE ,EMIT "You Win",EMIT "You Lose"
```

Let's go through every bytecode in order and discuss their significance and implementation.

BLK_NO - This is a custom bytecode that fetches the current block number from the local substrate instance that is running and pushes it to the stack.

ICONST_2 - This is a JVM bytecode which pushes the constant number *2* into the stack.

IREM - We use the JVM bytecode that performs a modulus operation (%) to the top two elements on the stack, the block number and the constant *2*, and pushes the result into the stack.

IF_ICMPNE - This JVM bytecode executes an if statement that compares if the modulus result is equal to zero. We have modified the approach of the bytecode implementation that will be discussed in the upcoming section 2.3.

EMIT "You Win" - This is a custom bytecode which loads the string *"You Win"* into the stack and breaks the loop if the *if* condition is satisfied. Then the top element in the stack is broadcasted through an event.

EMIT "You Lose" - This is a custom bytecode which loads the string *"You Lose"* onto the stack and broadcasted as an event. This block will only be executed when the *if* condition is false.

Please use the exact order and format of bytecodes while uploading to execute successfully.

2.3 Interpreter Implementation

Let us now discuss in detail the *execute()* function. This function takes in a smart contract address of type *Vec<u8>* type and executes the smart contract stored in that address. We first get the bytecodes present in the address from the *SC* map and store them in a local variable. We now split the bytecode into a list of bytecodes and push it into an array. Now we loop through the bytecode array, *opcodes*, and using *match* case statement we define the functionality of each and every bytecode.

```
pub fn execute(origin: OriginFor<T>, address: Vec<u8>) -> DispatchResult {
    let _who = ensure_signed(origin)?;
    let bytecode = SC::<T>::get(address).ok_or(Error::<T>::NoneValue)?;
    let mut output:String = "".to_string();
    let opcodes: Vec<&str> = bytecode.split(",").collect();
    let mut stack: Vec<String> = Vec::new();
    let mut bno: Option<u32> = Some(0);
    let mut ifeq: u32 = 0;
    for opcode in opcodes {
        match opcode {
            "BLK_NO" => {
                let current_block_number = <frame_system::Pallet<T>>::block_number();

                bno = TryInto::<u32>::try_into(current_block_number).ok();
                stack.push((bno).expect("REASON").to_string());
            },
            "ICONST_2" => {
                stack.push(2.to_string());
            }
            "IREM" => {
                let a = (stack.pop()).expect("REASON").parse::<i32>().unwrap();
                let b = (stack.pop()).expect("REASON").parse::<i32>().unwrap();
                stack.push((b%a).to_string());
            },
            "IF_ICMPNE " => {
                let rem = (stack.pop()).expect("REASON").parse::<i32>().unwrap();
                if rem == 0{
                    ifeq = 1;
                }
                else {
                    continue;
                }
            },
            "EMIT \"You Win\"" => {
                if ifeq == 1{
```

```
                    stack.push("You Win".to_string());
                    break;
                }
                else{
                    continue;
                }
            },
            "EMIT \"You Lose\"" => {
                stack.push("You Lose".to_string());
                break;
            },


            &_ => {continue;}
        }
    }
    output = stack.pop().expect("REASON").to_string();
    Self::deposit_event(Event::SCExecuted { output: output.to_string(), bno });

    Ok(())
}
```

For the implementation of *BLK_NO*, we get the local block number using the *frame_system* pallet and push it into the stack as a *String*. After pushing the constant number *2* onto the stack we calculate the modulus of the top two elements of the stack and push the result to the stack.

Now for the if statement, we use the same bytecode as the JVM but slightly modify the approach. As implementing a *goto* statement in rust is complicated (discussed in section 3), we perform the logic of the if statement and store the result in a local flag variable. We have implemented the interpreter in a way that if the *if* statement is satisfied the next bytecode is executed, else we skip the next bytecode. We then move to the *EMIT* bytecode. Here we load the string onto the stack, with respect to the if condition and its result. Then at last we pop the last element in the stack and broadcast it through an event to the blockchain.

3 Limitations

A custom substrate pallet that can store, retrieve and execute smart contracts on the ledger has been successfully created. We have managed to achieve all the main required specifications and functionalities. As a limitation, we may point out the implementation of the *IF_ICMPNE* JVM bytecode on our custom limited interpreter. We have basically hardcoded the logic and that is not the optimal way to achieve the functionality. Using the *goto* crate provided by the rust is recommended, but due to technical constraints between the crates, we weren't able to implement that. Some minor modifications in our interpreter implementation and some other count variables can improve the efficiency of our code. Adding the implementation of other JVM bytecodes can be the next step to improve the execution engine.