Actors in blockchain

Previously we were talking about actors mostly in the abstraction of any blockchain specific terms. However before we would dive into the code, we need to establish some common language, and to do so we would look at contracts from the perspective of external users, instead of their implementation.

In this part, I would use thewasmd binary to communicate with cliffnet testnet. To properly set it up, check the Setting up Environment .

Blockchain as a database

It is kind of starting from the end, but I would start with the state part of the actor model. Relating to traditional systems, there is one particular thing I like to compare blockchain with - it is a database.

Going back to the previous section we learned, that the most important part of a contract is its state. Manipulating the state is the only way to persistently manifest work performed to the world. But What is the thing which purpose is to keep the state? It is a database!

So here is my (as contract developer) point of view on contracts: it is distributed database, with some magical mechanisms to make it democratic. Those "magical mechanisms" are crucial for BC's existence and they make they are reasons why even use blockchain, but they are not relevant from the contract creator's point of view - for us, everything matter is the state.

But you can say: what about the financial part?! Isn't blockchain (wasmd in particular) the currency implementation? With all of those gas costs, sending funds seems very much like a money transfer, not database updates. And yes, you are kind of right, but I have a solution for that too. Just imagine, that for every native token (by "native tokens" we meant tokens handled directly by blockchain, in contradiction to for example cw20 tokens) there is a special database bucket (or table if you prefer) with mapping of address to how much of a token the address possesses. You can query this table (querying for token balance), but you cannot modify it directly. To modify it you just send a message to a special build-in bank contract. And everything is still a database.

But if blockchain is a database, then where smart contracts are stored? Obviously - in the database itself! So now imagine another special table - this one would contain a single table of code-ids mapped to blobs of wasm binaries. And again - to operate on this table, you use "special contract" which is not accessible from another contract, but you can use it viawasmd binary.

Now there is a question - why do I even care about BC being a DB? So the reason is that it makes reasoning about everything in blockchain very natural. Do you remember, that every message in the actor model is transactional? It perfectly matches traditional database transactions (meaning: every message starts a new transaction)! Also when we would later talk about migrations, it would turn out, that migrations in CosmWasm are very much equivalents of schema migrations in traditional databases.

So the thing to remember - blockchain is very similar to a database, having some specially reserved tables (like native tokens, code repository), with a special bucket created for every contract. A contract can look at every table in every bucket in the whole blockchain, but it can modify the only one he created.

Compile the contract

I will not go into the code for now, but to start with something we need compiled contract binary. Thecw4-group contract fromcw-plus is simple enough to work with, for now, so we will start with compiling it. Start with cloning the repository:

git clone git@github.com:CosmWasm/cw-plus.git Then go tocw4-group contract and build it:

cd cw-plus/contracts/cw4-group cargo wasm Your final binary should be located in thecw-plus/target/wasm32-unknown-unknown/release folder (cw-plus being where you cloned your repository).

Contract code

When the contract binary is built, the first interaction with CosmWasm is uploading it to the blockchain (assuming you have your wasm binary in the working directory):

wasmd tx wasm store ./cw4-group.wasm --from wallet TXFLAG -y As a result of such an operation you would get json output like this:

.. logs: .. - events: .. - attributes: - key: code_id value: "1069" type: store_code I ignored most of not fields as they are not relevant for now - what we care about is the event emitted by blockchain with information aboutcode_id of stored contract - in my case the contract code was stored in blockchain under id of1069 . I can now look at the code by querying for it:

wasmd query wasm code 1069 code.wasm And now the important thing - the contract code is not an actor. So what is a contract code? I think that the easiest way to think about that is aclass or atype in programming. It defines some stuff about what can be done, but the class itself is in most cases not very useful unless we create an instance of a type, on which we can call class methods. So now let's move forward to instances of such contract classes.

Contract instance

Now we have a contract code, but what we want is an actual contract itself. To create it, we need to instantiate it. Relating to analogy to programming, instantiation is calling a constructor. To do that, I would send an instantiate message to my contract:

wasmd tx wasm instantiate 1069

'{"members": []}' --from wallet --label "Group 1" --no-admin TXFLAG -y What I do here is creating a new contract and immediately call theInstantiate message on it. The structure of such a message is different for every contract code. In particular, thecw4-group Instantiate message contains two fields:

- members
- field which is the list of initial group members
- optionaladmin
- field which defines an address who can add or remove
- group member

In this case, I created an empty group with no admin - so which could never change! It may seem like a not very useful contract, but it serves us as a contract example.

As the result of instantiating I got result:

.. logs: .. - events: .. - attributes: - key: _contract_address value: wasm1u0grxl65reu6spujnf20ngcpz3jvjfsp5rs7lkavud3rhppnyhmqqnkcx6 - key: code_id value: "1069" type: instantiate As you can see we again look atlogs[].events[] field, looking for interesting event and extracting information from it - it is the common case. I will talk about events and their attributes in the future but in general it is a way to notify the world that something happened. Do you remember the KFC example? If a waiter is serving our dish, he would put a tray on the bar, and she would yell (or put on the screen) the order number - this would be announcing an event, so you know some summary of operation, so you can go and do something useful with it.

So what use can we do with the contract? We obviously can call it! But first I want to tell you about addresses.

Addresses in CosmWasm

Address in CosmWasm is a way to refer to entities in the blockchain. There are two types of addresses: contract addresses, and non-contracts. The difference is, that you can send messages to contract addresses, as there is some smart contract code associated with them, and non-contracts are just users of the system. In an actor model, contract addresses represent actors, and non-contracts represent clients of the system.

When operating with blockchain usingwasmd , you also have an address - you got one when you added the key towasmd :

add wallets for testing

wasmd keys add wallet3 - name: wallet3 type: local address: wasm1dk6sq0786m6ayg9kd0ylgugykxe0n6h0ts7d8t pubkey: '{"@type":"/cosmos.crypto.secp256k1.PubKey","key":"Ap5zuScYVRr5Clz7QLzu0CJNTg07+7GdAAh3uwgdig2X"}' mnemonic: "" You can always check your address:

wasmd keys show walle - name: wallet type: local address: wasm1um59mldkdj8ayl5gknp9pnrdlw33v40sh5l4nx pubkey: '{"@type":"/cosmos.crypto.secp256k1.PubKey","key":"A5bBdhYS/4qouAfLUH9h9+ndRJKvK0co31w4lS4p5cTE"}' mnemonic: "" Having an address is very important because it is requirement to being able to call anything. When we send a message to a contract it always knows the address who sends this message so it can identify it - not to mention, that this sender is an address which would play a gas cost.

Querying the contract

So we have our contract, let try to do something with it - query would be the easiest thing to do. Let's do it:

wasmd query wasm contract-state smart wasm1u0grxl65reu6spujnf20ngcpz3jvjfsp5rs7lkavud3rhppnyhmqqnkcx6 '{ "list_members": {} }' data: members: [ ] Thewasm... string is the contract address, and you have to substitute it with your contract address.{ "list_members": {} } is query message we send to contract. Typically CW smart contract queries are in the form of a single JSON object, with one field: the query name (list_members in our case). The value of this field is another

object, being query parameters - if there are any.list_members query handles two parameters:limit , andstart_after , which are both optional and which support result pagination. However in our case of an empty group they don't matter.

The query result we got is in human-readable text form (if we want to get the JSON from - for example, to process it further withjq , just pass the-o json flag). As you can see response contains one field:members which is an empty array.

So can we do anything more with this contract? Not much. But let's try to do something with a new one!

Executions to perform some actions

The problem with our previous contract is, that forcw4-group contract, the only one who can perform executions on it, is an admin, but our contract doesn't have one. This is not a true for every smart contract, but it is a nature of this one.

So let's make a new group contract, but this time we would make ourselves an admin. First, check our wallet address:

wasmd keys show wallet And instantiate new group contract - this time with proper admin:

wasmd tx wasm instantiate 1069

'{"members": [], "admin": "wasm1um59mldkdj8ayl5gknp9pnrdlw33v40sh5l4nx"}' --from wallet --label "Group 1" --no-admin TXFLAG -y .. logs: - events: .. - attributes: - key: _contract_address value: wasm1n5x8hmstlzdzy5jxd70273tuptr4zsclrwx0nsqv7qns5gm4vraqeam24u - key: code_id value: "1069" type: instantiate You may ask, why do we pass some kind of--no-admin flag, if we just said, we want to set an admin to the contract? The answer is sad and confusing, but... it is a different admin. The admin we want to set is one checked by the contract itself and managed by him. The admin which is declined with--no-admin flag, is a wasmd-level admin, which can migrate contract. You don't need to worry about the second one at least until you will learn about contracts migrations - until then you can always pass the--no-admin flag to the contract.

Now let's query our new contract for the member's list:

wasmd query wasm contract-state smart wasm1n5x8hmstlzdzy5jxd70273tuptr4zsclrwx0nsqv7qns5gm4vraqeam24u '{ "list_members": {} }' data: members: [ ] Just like before - no members initially. Now check an admin:

wasmd query wasm contract-state smart wasm1n5x8hmstlzdzy5jxd70273tuptr4zsclrwx0nsqv7qns5gm4vraqeam24u '{ "admin": {} }' data: admin: wasm1um59mldkdj8ayl5gknp9pnrdlw33v40sh5l4nx So there is an admin, seems like the one we wanted to have there. So now we would add someone to the group - may be ourselves?

wasmd tx wasm execute wasm1n5x8hmstlzdzy5jxd70273tuptr4zsclrwx0nsqv7qns5gm4vraqeam24u '{ "update_members": { "add": [{ "addr": "wasm1um59mldkdj8ayl5gkn p9pnrdlw33v40sh5l4nx", "weight": 1 }], "remove": [] } }' --from wallet TXFLAG -y The message for modifying the members isupdate_members and it has two fields: members to remove, and members to add. Members to remove are just addresses. Members to add has a bit more complex structure: they are records with two fields: address and weight. Weight is not relevant for us now, it is just metadata stored with every group member - for us, it would be always 1.

Let's query the contract again to check if our message changed anything:

wasmd query wasm contract-state smart wasm1n5x8hmstlzdzy5jxd70273tuptr4zsclrwx0nsqv7qns5gm4vraqeam24u '{ "list_members": {} }' data: members: - addr: wasm1um59mldkdj8ayl5gknp9pnrdlw33v40sh5l4nx weight: 1 As you can see, the contract updated its state. This is basically how it works - sending messages to contracts causes them to update the state, and the state can be queried at any time. For now, to keep things simple we were just interacting with the contract directly bywasmd , but as described before - contracts can communicate with each other. However, to investigate this we need to understand how to write contracts. Next time we would look at the contract structure and we will map it part by part to what we learned until now. Previous Idea behind an Actor Model Next Smart contract as an actor * Blockchain as a database * Compile the contract * Contract code * Contract instance * Addresses in CosmWasm * Querying the contract * Executions to perform some actions