Idea behind an Actor Model

The actor model is the solution to the problem of communication between smart contracts. Let's take a look at the reasons why this particular solution is chosen in CosmWasm, and what are the consequences of that.

The problem

Smart contracts can be imagined as sandboxed microservices. Due to SOLID principles, it is valuable to split responsibilities between entities. However, to split the work between contracts themselves, there is a need to communicate between them, so if one contract is responsible for managing group membership, it is possible to call its functionality from another contract.

The traditional way to solve this problem in SW engineering is to model services as functions that would be called with some RPC mechanism, and return its result as a response. Even though this approach looks nice, it creates sort of problems, in particular with shared state consistency.

The other approach which is far more popular in business level modelling is to treat entities as actors, which can perform some task, but without interrupting it with calls to other contracts. Any calls to other contracts can be called only after the whole execution is performed. When "subcall" would be finished, it will call the original contract back.

This solution may feel unnatural, and it requires to kind of different design solutions, but it turns out to work pretty well for smart contracts execution. I will try to explain how to reason about it, and how it maps to contract structure step by step.

The Actor

The most important thing in the whole model is an Actor itself. So what is this? The Actor is a single instantiation of a contract, which can perform several actions. When the actor finishes his job, he prepares a summary of it, which includes the list of things that have to be done, to complete the whole scheduled task.

An example of an actor is the Seller in the KFC restaurant. The first thing you do is order your BSmart, so you are requesting action from him. So from the system user, you can think about this task as "sell and prepare my meal", but the action performed by the seller is just "Charge payment and create order". The first part of this operation is to create a bill and charge you for it, and then it requests the Sandwich and Fries to be prepared by other actors, probably chefs. Then when the chef is done with his part of the meal, he checks if all meals are ready. If so, it calls the last actor, the waiter, to deliver the food to you. At this point, you can receive your delivery, and the task is considered complete.

The above-described workflow is kind of simplified. In particular - in a typical restaurant, a waiter would observe the kitchen instead of being triggered by a chef, but in the Actor model, it is not possible. Here, entities of the system are passive and cannot observe the environment actively - they only react to messages from other system participants. Also in KFC, the seller would not schedule subtasks for particular chefs, instead, he would leave tasks to be taken by them, when they are free. It is not the case, because as before - chefs cannot actively listen to the environment. However, it would be possible to create a contract for being a chefs dispatcher which would collect all orders from sellers, and balance them across chefs for some reason.

The Action

Actors are the model entities, but to properly communicate with them, we need some kind of protocol. Every actor is capable of performing several actions. In my previous KFC example, the only action seller can do is "Charge payment and create order". However, it is not always the case - our chefs were proficient with performing both "Prepare fries" and "Prepare Sandwich" actions - and also many more.

So when we want to do something in an actor system, we schedule some action to the actor being the closest to us, very often with some additional parameters (as we can pick if we want to exchange fries with salad).

However, naming the action after the exact thing which happened in the very contract would be misleading. Take a look at the KFC example once again. As I mentioned, the action performed by a seller is "Charge payment and create order". The problem is, that for the client which schedules this action, it doesn't matter what exactly is the responsibility of the actor himself - what the client is scheduling is "Prepare Meal" with some description of what exactly to prepare. So we can say, that the action is the thing performed by the contract itself, plus all the sub-actions it schedules.

Multi-stage Actions

So as the whole idea makes some sense, there is the problem created by the actor model: what if I want to perform some action in my contract, but to completely finalize some steps, the contract has to make sure that some sub-action he scheduled are finished?

Imagine, that in the previous KFC situation there is no dedicated Waiter, instead the Seller is serving you a meal when the

Chefs finished their job.

This kind of pattern is so important and common, that in CosmWasm we developed a special way to handle it, which is dedicatedReply action.

So when Seller is scheduling actions for chefs, he assigns some number to this action (like order id) and passes it to chefs. He also remembers how many actions he scheduled for every order id. Now every time chef is finished with his action, he would call the specialReply action on Seller, in which he would pass back the order id. Then Seller would decrease the number of actions left for this order, and if it reached zero, he would serve a meal.

Now you can say, that theReply action is completely not needed, as Chefs could just schedule any arbitrary action on Seller, likeServe , why there is the specialReply for? The reason is abstraction and reusability. The Chefs task is to prepare a meal, and that is all. There is no reason for him to know why he is even preparing Fries - if it is part of the bigger task (like order for a client), or the seller is just hungry. It is possible, that not only the seller is eligible to call the chef for food - possibly any restaurant employee can do that just for themselves. Therefore, we need a way, to be able to react to an actor finishing his job in some universal way, to handle this situation properly in any context.

It is worth noting, that theReply can contain some additional data. The id assigned previously is the only required information in theReply call, but the actor can pass some additional data -events emitted, which are mostly metadata (to be observed by non-blockchain applications mostly), and any arbitrary data it wants to pass.

State

Up until this point, we were considering actors as entities performing some job, like preparing the meal. If we are considering computer programs, such a job would be to show something on the screen, maybe print something. This is not the case with Smart Contracts. The only thing which can be affected by the Smart Contract is their internal state. So the state is arbitrary data that is kept by the contract. Previously in the KFC example I mentioned, that Seller is keeping in mind how many actions he scheduled for chefs are not yet finished - this number is part of the Seller's state.

To give a more realistic example of a contract state, let's think about a more real-life Smart Contract than the restaurant. Let's imagine we want to create our currency - maybe we want to create some smart contracts-based market for some MMORPG game. So we need some way, to be able to at least transfer currency between players. We can do that, by creating the contract we would callMmoCurrency , which would support theTransfer action to transfer money to another player. Then what would be the state of such a contract? It would be just a table mapping player names to the amount of currency they own. The contract we just invented exists in CosmWasm examples, and it is called thecw20-base contract (it is a bit more complicated, but it is its core idea).

And now there is a question - how is this helpful to transfer currency if I cannot check how much of it I own? It is a very good question, and the answer to that is simple - the whole state of every contract in our system is public. It is not universal for every Actor model, but it is how it works in CosmWasm, and it is kind of forced by the nature of blockchain. Everything happening in blockchain has to be public, and if some information should be hidden, it has to be stored indirectly.

There is one very important thing about the state in CosmWasm, and it is the state being transactional. Any updates to the state are not applied immediately, but only when the whole action succeeds. It is very important, as it guarantees that if something goes wrong in the contract, it is always left in some proper state. Let's consider ourMmoCurrency case. Imagine, that in theTransfer action we first increase the receiver currency amount (by updating the state), and only then do we decrease the sender amount. However, before decreasing it, we need to check if a sender possesses enough funds to perform the transaction. In case we realize, that we cannot do it, we don't need to do any rolling back by hand - we would just return a failure from the action execution, and the state would not be updated. So when in contract state is updated, it is just a local copy of this state being altered, but the partial changes would never be visible by other contracts.

Queries

There is one building block in the CosmWasm approach to the Actor model, which I didn't yet cover. As I said, the whole state of every contract is public and available for everyone to look at. The problem is, that this way of looking at state is not very convenient - it requires users of contracts to know its internal structure, which kind of violates the SOLID rules (Liskov substitution principle in particular). If for example contract is updated and its state structure changes a bit, another contract looking at its state would just nevermore work. Also, it is often the case, that the contract state is kind of simplified, and information that is relevant to the observer would be calculated from the state.

This is where queries come into play. Queries are the type of messages to contract, which does not perform any actions, so do not update any state, but can return an answer immediately.

In our KFC comparison, the query would be if Seller goes to Chef to ask "Do we still have pickles available for our cheeseburgers"? It can be done while operating, and response can be used in it. It is possible because queries can never update their state, so they do not need to be handled in a transactional manner.

However, the existence of queries doesn't mean, that we cannot look at the contract's state directly - the state is still public, and the technique of looking at them directly is calledRaw Queries . For clarity, non-raw queries are sometimes denoted

asSmart Queries .

Wrapping everything together - transactional call flow

So we touched on many things here, and I know it may be kind of confusing. Because of that, I would like to go through some more complicated calls to CosmWasm contract to visualize what the "transactional state" means.

Let's imagine two contracts:

1. TheMmoCurrency
2. contract mentioned before, which can performTransfer
3. action, allows transferring someamount
4. of currency to somereceiver
5. .
6. TheWarriorNpc
7. contract, which would have some amount of our currency,
8. and he would be used by our MMO engine to pay the reward out for some
9. quest player could perform. It would be triggered byPayout
10. action,
11. which can be called only by a specific client (which would be our game
12. engine).

Now here is an interesting thing - this model forces us, to make our MMO more realistic in terms of the economy that we traditionally see - it is becauseWarriorNpc has some amount of currency, and cannot create more out of anything. It is not always the case (the previously mentionedcw20 has a notion of Minting for this case), but for sake of simplicity let's assume this is what we want.

To make the quest reasonable for longer, we would make a reward for it to be always between1 mmo and100 mmo , but it would be ideally15% of what Warrior owns. This means, that the quest reward decreases for every subsequent player, until Warrior would be broke, left with nothing, and will no longer be able to payout players.

So what would the flow look like? The first game would send aPayout message to theWarriorNpc contract, with info on who should get the reward. Warrior would keep track of players who fulfilled the quest, to not pay out the same person twice - there would be a list of players in his state. First, he would check the list looking for players to pay out - if he is there, he would finish the transaction with an error.

However, in most cases the player would not be on the list - so thenWarriorNpc would add him to the list. Now the Warrior would finish his part of the task, and schedule theTransfer action to be performed byMmoCurrency .

But there is the important thing - becauseTransfer action is actually the part of the biggerPayout flow, it would not be executed on the original blockchain state, but on the local copy of it, which the player's list is already applied to. So if theMmoCurrency would for any reason takes a look atWarriorNpc internal list, it would be already updated.

NowMmoCurrency is doing its job, updating the state of Warrior and player balance (note, that our Warrior is here just treated as another player!). When it finishes, two things may happen:

1. There was an error - possibly Warrior is out of cash, and it can nevermore
2. pay for the task. In such case, none of the changes - neither updating the list
3. of players succeeding, nor balance changes are not applied to the original
4. blockchain storage, so they are like never happened. In the database world, it
5. is denoted as rolling back the transaction.
6. Operation succeed - all changes on the state are now applied to blockchain,
7. and any further observation ofMmoCurrency
8. orWarriorNpc
9. by external
10. the world would see updated data.

There is one problem - in this model, our list is not a list of players who fulfilled the quest (as we wanted it to be), but the list of players paid out (as in transfer failure, the list is not updated). We can do better.

Different ways of handling responses

Note, that we didn't mention aReply operation at all. So why it was not called byMmoCurrency onWarriorNpc ? The reason is, that this operation is optional. When scheduling sub-actions on another contract we may choose whenReply how the result should be handled:

1. Never callReply
2. , action fails if sub-message fails
3. CallReply

4. on success
5. CallReply
6. on failure
7. Always callReply

So if we do not requestReply to be called by subsequent contract, it will not happen. In such a case if a sub-call fails, the whole transaction is rolled back - sub-message failure transitively causes the original message failure. It is probably a bit complicated for now, but I promise it would be simple when you would do some practice with that.

When handling the reply, it is important to remember, that although changes are not yet applied to the blockchain (the transaction still can be failed), the reply handler is already working on the copy of the state with all changes made by sub-message so far applied. In most cases, it would be a good thing, but it has a tricky consequence - if the contract is calling itself recursively, it is possible that subsequent call overwrote things set up in the original message. It rarely happens, but may need special treatment in some cases - for now I don't want to go deeply into details, but I want you to remember about what to expect after state in the actor's flow.

Now let's take a look at handling results with2 -4 options. It is actually interesting, that using2 , even if the transaction is performed by sub-call succeed, we may now take a look at the data it returned withReply , and on its final state after it finished, and we can still decide, that action as a whole is a failure, in which case everything would be rolled back - even currency transfer performed by external contract.

In our case, an interesting option is3 . So if the contract would callReply on failure, we can decide to claim success, and commit a transaction on the state if the sub call failed. Why it may be relevant for us? Possibly because our internal list was supposed to keep the list of players succeeding with the quest, not paid out! So if we have no more currency, we still want to update the list!

The most common way to use the replies (option2 in particular) is to instantiate another contract, managed by the called one. The idea is, that in those use cases, the creator contract wants to keep the address of created contract in its state. To do so it has to create anInstantiate sub-message, and subscribe for its success response, which contains an address of freshly created contract.

In the end, you can see, that performing actions in CosmWasm is built with hierarchical state change transactions. The sub-transaction can be applied to the blockchain only if everything succeeds, but in case that sub-transaction failed, only its part may be rolled back, end other changes may be applied. It is very similar to how most database systems work.

Conclusion

Now you have seen the power of the actor model to avoid reentrancy, properly handle errors, and safely sandbox contracts. This helps us provide the solid security guarantees of the CosmWasm platform. Let’s get started playing around with real contracts in thewasmd blockchain. Previous Integration Next Actors in blockchain * The problem * The Actor * The Action * Multi-stage Actions * State * Queries * Wrapping everything together - transactional call flow* Different ways of handling responses * Conclusion