Actor Model for Contract Calls

Theactor model is a design pattern, used to build reliable, distributed systems. The fundamental points are that eachActor has exclusive access to its own internal state and thatActors cannot call each other directly. Instead, they dispatch messages over someDispatcher (that maintains the state of the system and maps addresses to code and storage). Fundamentally theActor pattern can be encapsulated in this interface:

pub

trait

Actor

{ fn

handle (msgPayload :

& [u8])

->

Vec < Msg

    ; }

pub

struct

Msg

{ pub destination :

Vec < u8

    , pub payload :

Vec < u8

    , } This is the basic model that was used to model contracts in CosmWasm. You can see the same influence in the function:

pub

fn

handle < T :

Storage

    (store :

& mut

T , params :

Params , msg :

Vec < u8

)

->

Result < Response

    The response containsVec and a little metadata.store is access to the contract's internal state. Andparams is some global immutable context. So, just a little bit of syntax around the same design. From this basic design, a few other useful aspects can be derived:

- First, there is aloose coupling
- between Actors, limited to the format of the data packets (the recipient must

- support a superset of what you send). There are no complex API or function pointers to pass around. This is much like
- using REST or RPC calls as a boundary between services, which is a scalable way to compose systems from many vendors.
- Secondly, eachActor
- can effectively run on its own thread, with its own queue. This both enables concurrency (which
- we don't make use of in CosmWasm... yet), andserialized execution
- within each actor (which we do rely upon). This
- means that theHandle
- method above cannot be executed in the middle of a previously executedHandle
- call.Handle
- is a synchronous call and returns before theActor
- can process the next message. This feature is
- whatprotects us from reentrancy by design
- .

info The contract can access other contracts state directly - such technique is called raw querying. However, contract can never write to others contracts state. Another important aspect related to CosmWasm islocality . That is, actors can only communicate with other actorswhose address they previously received . We will go more into depth onnames and addresses on the next page, but the key point is that for two actors to communicate, an external message (from the contract creator, or potentially a user) must be sent to the actor. This is a flexible way to set up topologies in a distributed manner. The only thing that must be hard-coded is the data format to pass to such addresses. Once some standard interfaces are established (like ERC20, ERC721, ENS, etc), then we can support composability between large classes of contracts, with different backing codes, but sharing a common API.

Security Benefits

By enforcingprivate internal state , a given contract can guarantee all valid transitions in its internal state. This is in contrast to the capabilities model used in Cosmos SDK, where trusted modules are passed aStoreKey in their constructor, which allowsfull read and write access to the other module's storage . In the Cosmos SDK, we can audit the modules before calling them, and safely pass in such a powerful set of rights at compile time. However, there are no compile-time checks in a smart contract system and we need to produce stricter boundaries between contracts. This allows us to comprehensively reason over all possible transitions in a contract's state (and use quick-check-like methods to test it).

As mentioned above,serialized execution prevents all concurrent execution of a contract's code. This is like an automatic mutex over the entire contract code. This is exactly the issue that one of the most common Ethereum attacks, reentrancy, makes use of. Contract A calls into contract B, which calls back into contract A. There may be local changes in memory in contract A from the first call (eg. deduct a balance), which are not yet persisted, so the second call can use the outdated state a second time (eg. authorize sending a balance twice). By enforcing serialized execution, the contract will write all changes to storage before exiting and have a proper view when the next message is processed.

Atomic Execution

One problem with sending messages is atomically committing a state change over two contracts. There are many cases where we want to ensure that all returned messages were properly processed before committing our state. There are ideas like " three-phase-commit" is used in distributed databases, but since in the normal case, all actors are living in the same binary, we can handle this in theKeeper . Before executing a Msg that came from an external transaction, we create a SavePoint of the global data store, and pass in a subset to the first contract. We then execute all returned messages inside the same sub-transaction. If all messages succeed, then we can commit the sub-transaction. If any fails (or we run out of gas), we abort execution and roll back the state to before the first contract was executed.

This allows us to optimistically update code, relying on rollback for error handling. For example, if an exchange matches a trade between two "ERC20" tokens, it can make the offer fulfilled and return two messages to move token A to the buyer and token B to the seller. (ERC20 tokens use a concept of allowance, so the owner "allows" the exchange to move up to X tokens from their account). When executing the returned messages, it turns out that the buyer doesn't have sufficient token B (or provided an insufficient allowance). This message will fail, causing the entire sequence to be reverted. Transaction failed, the offer was not marked as fulfilled, and no tokens changed hands.

While many developers may be more comfortable thinking about directly calling the other contract in their execution path and handling the errors, you can achieve almost all the same cases with such anoptimistic update and return approach. And there is no room for making mistakes in the contract's error handling code.

Dynamically Linking Host Modules

The aspects oflocality andloose coupling means that we don't even need to link to other CosmWasm contracts. We can send messages to anything the Dispatcher has an address for. For example, we can return aSendMsg , which will be processed by the nativex/bank module in Cosmos SDK, moving native tokens. As we define standard interfaces for composability, we can define interfaces to call into core modules (bond and unbond your stake...), and then pass in the address to the native module in the contract constructor.

Inter Blockchain Messaging

Since the Actor model doesn't attempt to make synchronous calls to another contract but just returns a message "to be executed", it is a nice match for making cross-chain contract calls usingIBC . The only caveat here is that theatomic execution guarantee we provided above no longer applies here. The other call will not be called by the same dispatcher, so we need to store an intermediate state in the contract itself. That means a state that cannot be changed until the result of the IBC call is known, then can be safely applied or reverted.

For example, if we want to move tokens from chain A to chain B, we would first prepare a send:

1. Contract A reduces the token supply of the sender
2. Contract A creates an "escrow" of those tokens linked to the IBC message id, sender, and receiving chain.
3. Contract A commits state and returns a message to initiate an IBC transaction to chain B.
4. If the IBC send part fails, then the contract is atomically reverted as above.

After some time, a "success" or "error"/"timeout" message is returned from the IBC module to the token contract:

1. Contract A validates the message came from the IBC handler (authorization) and refers to a known IBC message ID it
2. has in escrow.
3. If it was a success, the escrow is deleted and the escrowed tokens are placed in an account for "Chain B" (meaning
4. that only a future IBC message from Chain B may release them).
5. If it was an error, the escrow is deleted and the escrowed tokens are returned to the account of the original sender.

You can imagine a similar scenario working for cases like moving NFT ownership, cross-chain staking, etc. We will expand on these possibilities and provide tooling to help make proper design once the IBC code in Cosmos SDK is stabilized (and included in a release), but the contract design is made with this in mind. Previous What are Multi-Chain Contracts?Next Names and Addresses * Security Benefits * Atomic Execution * Dynamically Linking Host Modules * Inter Blockchain Messaging