Contract Semantics

This document aims to clarify the semantics of how a CosmWasm contract interacts with its environment. There are two main types of actions:mutating actions, which receiveDepsMut and are able to modify the state of the blockchain, andquery actions, which are run on a single node with read-only access to the data.

Execution

In the section below, we will discuss how theexecute call works, but the same semantics apply to any othermutating action - instantiate ,migrate ,sudo , etc.

SDK Context

Before looking at CosmWasm, we should look at the (somewhat under-documented) semantics enforced by the blockchain framework we integrate with - theCosmos SDK . It is based upon theTendermint BFT Consensus Engine. Let us first look how they process transactions before they arrive in CosmWasm (and after they leave).

First, the Tendermint engine will seek 2/3+ consensus on a list of transactions to be included in the next block. This is donewithout executing them . They are simply subjected to a minimal pre-filter by the Cosmos SDK module, to ensure they are validly formatted transactions, with sufficient gas fees, and signed by an account with sufficient fees to pay it. Notably, this means many transactions that error may be included in a block.

Once a block is committed (typically every 5s or so), the transactions are then fed to the Cosmos SDK sequentially in order to execute them. Each one returns a result or error along with event logs, which are recorded in theTxResults section of the next block. TheAppHash (or merkle proof or blockchain state) after executing the block is also included in the next block.

The Cosmos SDKBaseApp handles each transaction in an isolated context. It first verifies all signatures and deducts the gas fees. It sets the "Gas Meter" to limit the execution to the amount of gas paid for by the fees. Then it makes an isolated context to run the transaction. This allows the code to read the current state of the chain (after the last transaction finished), but it only writes to a cache, which may be committed or rolled back on error.

A transaction may consist of multiple messages and each one is executed in turn under the same context and same gas limit. If all messages succeed, the context will be committed to the underlying blockchain state and the results of all messages will be stored in theTxResult . If one message fails, all later messages are skipped and all state changes are reverted. This is very important for atomicity. That means Alice and Bob can both sign a transaction with 2 messages: Alice pays Bob 1000 ATOM, Bob pays Alice 50 ETH, and if Bob doesn't have the funds in his account, Alice's payment will also be reverted. This is just like a DB Transaction typically works.

x/wasm is a custom Cosmos SDK module, which processes certain messages and uses them to upload, instantiate, and execute smart contracts. In particular, it accepts a properly signedMsgExecuteContract , routes it toKeeper.Execute , which loads the proper smart contract and callsexecute on it. Note that this method may either return a success (with data and events) or an error. In the case of an error here, it will revert the entire transaction in the block. This is the context we find ourselves in when our contract receives theexecute call.

Basic Execution

When we implement a contract, we provide the following entry point:

pub

fn

execute (deps :

DepsMut , env :

Env , info :

MessageInfo , msg :

ExecuteMsg ,)

->

Result < Response ,

ContractError

{

} WithDepsMut , this can read and write to the backingStorage , as well as use theApi to validate addresses, andQuery the state of other contracts or native modules. Once it is done, it returns eitherOk(Response) orErr(ContractError) . Let's examine what happens next:

If it returnsErr , this error is converted to a string representation (err.to_string()), and this is returned to the SDK module.All state changes are reverted andx/wasm returns this error message, which willgenerally (see submessage exception below) abort the transaction, and return this same error message to the external caller.

If it returnsOk , theResponse object is parsed and processed. Let's look at the parts here:

pub

struct

Response < T

=

Empty

where T :

Clone

+

fmt :: Debug

+

PartialEq

+

JsonSchema , { /// Optional list of "subcalls" to make. These will be executed in order /// (and this contract's subcall_response entry point invoked) /// *before* any of the "fire and forget" messages get executed. pub submessages :

Vec < SubMsg < T

, /// After any submessages are processed, these are all dispatched in the host blockchain. /// If they all succeed, then the transaction is committed. If any fail, then the transaction /// and any local contract state changes are reverted. pub messages :

Vec < CosmosMsg < T

, /// The attributes that will be emitted as part of a "wasm" event pub attributes :

Vec < Attribute

, pub data :

Option < Binary

, } In the Cosmos SDK, a transaction returns a number of events to the user, along with an optional data "result". This result is hashed into the next block hash to be provable and can return some essential state (although in general client apps rely on Events more). This result is more commonly used to pass results between contracts or modules in the sdk. Note that theResultHash includes only theCode (non-zero meaning error) andResult (data) from the transaction. Events and log are available via queries, but there are no light-client proofs possible.

If the contract setsdata , this will be returned in theresult field.attributes is a list of{key, value} pairs which will beappended to a default event . The final result looks like this to the client:

{ "type" :

"wasm" , "attributes" :

[{

"key" :

"contract_addr" ,

"value" :

"cosmos1234567890qwerty"

} , {

"key" :

"custom-key-1" ,

"value" :

"custom-value-1"

} , {

"key" :

"custom-key-2" ,

"value" :

"custom-value-2"

}] }

Dispatching Messages

Now let's move onto themessages field. Some contracts are fine only talking with themselves, such as a cw20 contract just adjusting its balances on transfers. But many want to move tokens (native or cw20) or call into other contracts for more complex actions. This is where messages come in. We returnCosmosMsg , which is a serializable representation of any external call the contract can make. It looks something like this (withstargate feature flag enabled):

pub

enum

CosmosMsg < T

=

Empty

where T :

Clone

+

fmt :: Debug

+

PartialEq

+

JsonSchema , { Bank (BankMsg) , /// This can be defined by each blockchain as a custom extension Custom (T) , Staking (StakingMsg) , Distribution (DistributionMsg) , Stargate

{ type_url :

String , value :

Binary , } , Ibc (IbcMsg) , Wasm (WasmMsg) , } If a contract returns two messages - M1 and M2, these will both be parsed and executed inx/wasm with the permissions of the contract (meaninginfo.sender will be the contract not the original caller). If they return success, they will emit a new event with the custom attributes, thedata field will be ignored, and any messages they return will also be processed. If they return an error, the parent call will return an error, thus rolling back state of the whole transaction.

Note that the messages are executeddepth-first . This means if contract A returns M1 (WasmMsg::Execute) and M2 (BankMsg::Send), and contract B (from theWasmMsg::Execute) returns N1 and N2 (eg.StakingMsg andDistributionMsg), the order of execution would beM1, N1, N2, M2 .

This may be hard to understand at first. "Why can't I just call another contract?", you may ask. However, we do this to

prevent one of most widespread and hardest to detect security holes in Ethereum contracts - reentrancy. We do this by following the actor model, which doesn't nest function calls, but returns messages that will be executed later. This means all state that is carried over between one call and the next happens in storage and not in memory. For more information on this design, I recommend you readour docs on the Actor Model .

Submessages

As of CosmWasm 0.14 (April 2021), we have added yet one more way to dispatch calls from the contract. A common request was the ability to get the result from one of the messages you dispatched. For example, you want to create a new contract withWasmMsg::Instantiate , but then you need to store the address of the newly created contract in the caller. Withsubmessages , this is now possible. It also solves a similar use-case of capturing the error results, so if you execute a message from eg. a cron contract, it can store the error message and mark the message as run, rather than aborting the whole transaction. It also allows for limiting the gas usage of the submessage (this is not intended to be used for most cases, but is needed for eg. the cron job to protect it from an infinite loop in the submessage burning all gas and aborting the transaction).

This makes use ofCosmosMsg as above, but it wraps it inside aSubMsg envelope:

pub

struct

SubMsg < T

=

Empty

where T :

Clone

+

fmt :: Debug

+

PartialEq

+

JsonSchema , { pub id :

u64 , pub msg :

CosmosMsg < T

, pub gas_limit :

Option < u64

, pub reply_on :

ReplyOn , }

pub

enum

ReplyOn

{ /// Always perform a callback after SubMsg is processed Always , /// Only callback if SubMsg returned an error, no callback on success case Error , /// Only callback if SubMsg was successful, no callback on error case Success , } What are the semantics of a submessage execution. First, we create a sub-transaction context around the state, allowing it to read the latest state written by the caller, but write to yet-another cache. Ifgas_limit is set, it is sandboxed to how much gas it can use until it aborts withOutOfGasError . This error is caught and returned to the caller like any other error returned from contract execution (unless it burned the entire gas limit of the transaction). What is more interesting is what happens on completion.

If it return success, the temporary state is committed (into the caller's cache), and theResponse is processed as normal (an event is added to the current EventManager, messages and submessages are executed). Once theResponse is fully processed, this may then be intercepted by the calling contract (forReplyOn::Always andReplyOn::Success). On an error,

the subcall will revert any partial state changes due to this message, but not revert any state changes in the calling contract. The error may then be intercepted by the calling contract (forReplyOn::Always andReplyOn::Error).In this case, the messages error doesn't abort the whole transaction

Handling the Reply

In order to make use ofsubmessages , the calling contract must have an extra entry point:

[entry_point]

pub

fn

reply (deps :

DepsMut , env :

Env , msg :

Reply)

->

Result < Response ,

ContractError

{

}

pub

struct

Reply

{ pub id :

u64 , /// ContractResult is just a nicely serializable version ofResult<SubcallResponse, String> pub result :

ContractResult < SubcallResponse

, }

pub

struct

SubcallResponse

{ pub events :

Vec < Event

, pub data :

Option < Binary

, } After thesubmessage is finished, the caller will get a chance to handle the result. It will get the originalid of the subcall so it can switch on how to process this, and theResult of the execution, both success and error. Note that it includes all events returned by the submessage, which applies to native sdk modules as well (like Bank) as well as the data returned from below. This and the original call id provide all context to continue processing it. If you need more state, you must save some local context to the store (under theid) before returning thesubmessage in the originalexecute , and load it inreply . We explicitly prohibit passing information in contract memory, as that is the key vector for reentrancy attacks, which are a large security surface area in Ethereum.

Thereply call may returnErr itself, in which case it is treated like the caller errored, and aborting the transaction. However, on successful processing,reply may return a normalResponse , which will be processed as normal - events added to the EventManager, and allmessages andsubmessages dispatched as described above.

The onecritical difference withreply , is that wedo not drop data . Ifreply returnsdata: Some(value) in theResponse object, we will overwrite thedata field returned by the caller. That is, ifexecute returnsdata: Some(b"first thought") and thereply (with all the extra information it is privy to) returnsdata: Some(b"better idea") , then this will be returned to the caller ofexecute (either the client or another transaction), just as if the originalexecute and returneddata: Some(b"better idea") . Ifreply returnsdata: None , it will not modify any previously set data state. If there are multiple submessages all setting this, only the last one is used (they all overwrite any previousdata value). As a consequence, you can usedata: Some(b"") to clear previously set data. This will be represented as a JSON string instead ofnull and handled as any otherSome value.

Order and Rollback

Submessages (and their replies) are all executed before anymessages . They also follow thedepth first rules as withmessages . Here is a simple example. Contract A returns submessages S1 and S2, and message M1. Submessage S1 returns message N1. The order will be:S1, N1, reply(S1), S2, reply(S2), M1

Please keep in mind that submessageexecution andreply can happen within the context of another submessage. For examplecontract-A--submessage --> contract-B--submessage --> contract-C . Thencontract-B can revert the state forcontract-C and itself by returningErr in the submessagereply , but not revert contract-A or the entire transaction. It just ends up returningErr to contract-A'sreply function.

Note that errors are not handled withReplyOn::Success , meaning, in such a case, an error will be treated just like a normalmessage returning an error. This diagram may help explain. Imagine a contract returned two submesssage - (a) withReplyOn::Success and (b) withReplyOn::Error :

processing a) processing b) reply called may overwrite result from reply note ok ok a) a) returns success err err none none returns error (abort parent transaction) err ok none none returns error (abort parent transaction) ok err a)b) a)b) if both a) and b) overwrite, only b) will be used

Query Semantics

Until now, we have focused on theResponse object, which allows us to execute code in other contracts via the actor model. That is, each contract is run sequentially, one after another, and no nested calls are possible. This is essential to avoid reentrancy, which is when calling into another contract can change my state while I am in the middle of a transaction.

However, there are many times we need access to information from other contracts in the middle of processing, such as determining the contract's bank balance before sending funds. To enable this, we have exposed theread only Querier to enablesynchronous calls in the middle of the execution. By making it read-only (and enforcing that in the VM level), we can prevent the possibility of reentrancy, as the query cannot modify any state or execute our contract.

When we "make a query", we serialize aQueryRequest struct that represents all possible calls, and then pass that over FFI to the runtime, where it is interpretted in thex/wasm SDK module. This is extensible with blockchain-specific custom queries just likeCosmosMsg accepts custom results. Also note the ability to perform raw protobuf "Stargate" queries:

pub

enum

QueryRequest < C :

CustomQuery

{ Bank (BankQuery) , Custom (C) , Staking (StakingQuery) , Stargate

{ /// this is the fully qualified service path used for routing, /// eg. custom/cosmos_sdk.x.bank.v1.Query/QueryBalance path :

String , /// this is the expected protobuf message type (not any), binary encoded data :

Binary , } , Ibc (IbcQuery) , Wasm (WasmQuery) , } While this is flexible and needed encoding for the cross-language representation, this is a bit of mouthful to generate and use when I just want to find my bank balance. To help that, we often useQuerierWrapper , which wraps aQuerier and exposes a lot of convenience methods that just useQueryRequest andQuerier.raw_query under the hood.

You can read a longer explanation of theQuerier design in our docs . Previous Comparison with Solidity Contracts Next Messages * Execution * * SDK Context * * Basic Execution * * Dispatching Messages * * Submessages * Query Semantics