Area-52 Workshop 4

IBC and CosmWasm

Agenda

- Intro
  - IBC
  - CosmWasm
- How to
- Examples
- Resources

IBC

- The Inter-Blockchain Communication Protocol (IBC) handles authentication and transport of data between blockchains.

- The IBC protocol provides a permissionless way for relaying data packets between blockchains.

- The security of IBC reduces to the security of the participating chains.

- With cross-chain communication via IBC, a decentralized network of independent and interoperable chains exchanging information and assets is possible.

IBC

- If all relayers were acting maliciously, the packets sent would get rejected. Even if liveness would be affected, security would not be.

- Just spinning up a non-malicious relayer to relay packets (with correct proofs) would be effective, as relaying is permissionless.

IBC

IBC works by acknowledging packets through channels that are uniquely identified by the (channelID, portID) tuple

IBC/TAO: Standards defining the Transport, Authentication, and Ordering of packets, i.e. the infrastructure layer (Core, Client, Relayer).

IBC/APP: Standards defining the application handlers for the data packets being passed over the transport layer (ICS-20, ICS-721,ICS-27..).

IBC

Just few considerations:

If tokens were sent across a different channel, the denomination would be different, as the IBC denom is defined by: **ibc/<hash of the channel-id & port-id>.**

Usually the IBC-enabled CosmWasm smart contract would need to be deployed on two chains

CosmWasm

WASM

- WASM is a binary instruction format, designed as a portable compilation target for diverse programming languages.
- Smart contracts can be written in any programming language that compiles to WASM

CosmWasm

- CosmWasm provides a runtime for WebAssembly smart contracts
- The most common language to write smart contracts in CosmWasm is by using Rust.

CosmWasm

- Smart contracts in CosmWasm are WASM bytes stored on-chain at a specific identifier (**codeId)**.
- Once the **codeId** has been instantiated, it gets a contract address on the blockchain.
- Smart contracts and modules can interact with each other.

CosmWasm

```
archway new
archway config
archway build

cargo wasm

archway build --optimize
```

Setting up the project and creating the WASM executables

```
archway store
archway instantiate
Archway metadata
```

Deploy the smart contract

CosmWasm

```
archway instantiate --args
'{"count":0}'
```

Example of how to instantiate the smart contract

```
archway query contract-state smart
--args '{"get_count": {}}'
```

Examples on how to interact with the smart contract

```
archway tx --args '{"increment": {}}'
```

CosmWasm

Serialized execution prevents all concurrent execution of a contract's code, so that the contract will write all changes to storage before exiting and have a proper view when the next message is processed.

The way that CosmWasm is built, resembles having an automatic mutex over the entire contract code. This prevents reentrancy attacks, which are the most common attack vectors for smart contracts

CosmWasm messages are "fire-and-forget": you won't need to await a promise and worry about race conditions and reentrancy attacks.

The actor model is a design pattern where each Actor has exclusive access to its own internal state and that Actors cannot call each other directly.

Instead, they dispatch messages over some Dispatcher (that maintains the state of the system and maps addresses to code and storage).

CosmWasm

If we want to move tokens from chain A to chain B, the flow would be:

- Contract A reduces the token supply of the sender
- Contract A creates an "escrow" of those tokens linked to the IBC message id, sender, and receiving chain.
- Contract A commits state and returns a message to initiate an IBC transaction to chain B.
- If the IBC send part fails, then the contract is atomically reverted
- After some time, a "success" or "error"/"timeout" message is returned from the IBC module to the token contract:
- Contract A validates the message came from the IBC handler and refers to a known IBC message ID it has in escrow.
- If it was a success, the escrow is deleted and the escrowed tokens are placed in an account for "Chain B" (meaning that only a future IBC message from Chain B may release them).
- If it was an error, the escrow is deleted and the escrowed tokens are returned to the account of the original sender.

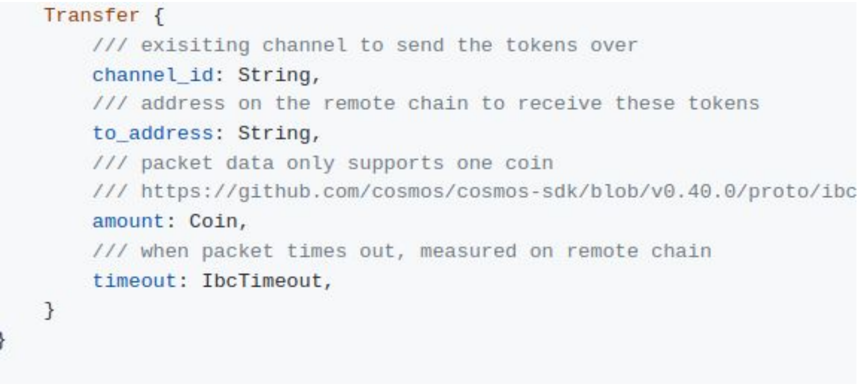
How to use IBC with CosmWasm

Few ways to use IBC with CosmWasm:

- Send tokens to another chain on a pre-established ICS20 channel.
- Implement custom ICS protocols inside the contract. (for example using the cw20-ics20 contract
- ICS-NFT transfer

Using an ICS channel

`CosmosMsg::Ibc(IbcMsg::Transfer{})`

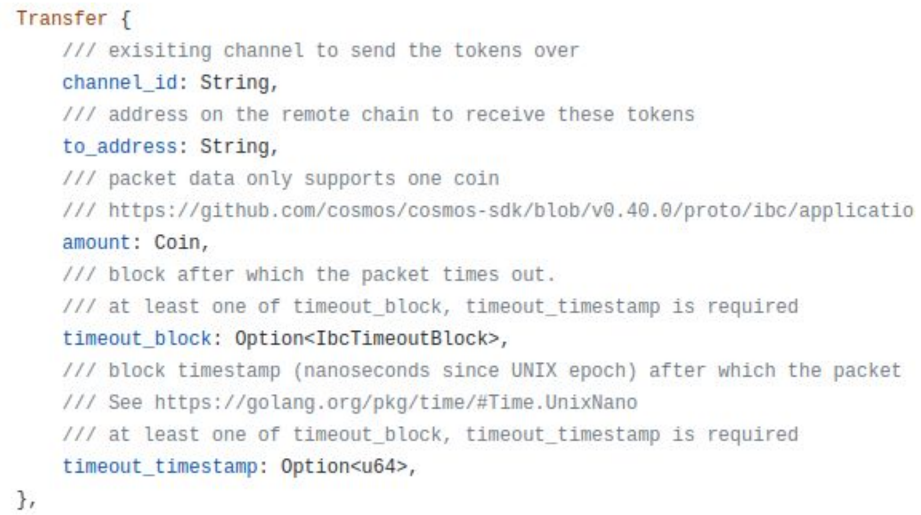


```
    Transfer {
        /// exisiting channel to send the tokens over
        channel_id: String,
        /// address on the remote chain to receive these tokens
        to_address: String,
        /// packet data only supports one coin
        /// https://github.com/cosmos/cosmos-sdk/blob/v0.40.0/proto/ibc
        amount: Coin,
        /// when packet times out, measured on remote chain
        timeout: IbcTimeout,
    }
}
```

In addition to the info you need in `BankMsg::Send`, you need to define:

- the `channel` to send upon
- The timeout specified either in block height or block time (or both).

`CosmosMsg::Bank(BankMsg::Send{})`



```
Transfer {
    /// exisiting channel to send the tokens over
    channel_id: String,
    /// address on the remote chain to receive these tokens
    to_address: String,
    /// packet data only supports one coin
    /// https://github.com/cosmos/cosmos-sdk/blob/v0.40.0/proto/ibc/applicatio
    amount: Coin,
    /// block after which the packet times out.
    /// at least one of timeout_block, timeout_timestamp is required
    timeout_block: Option<IbcTimeoutBlock>,
    /// block timestamp (nanoseconds since UNIX epoch) after which the packet
    /// See https://golang.org/pkg/time/#Time.UnixNano
    /// at least one of timeout_block, timeout_timestamp is required
    timeout_timestamp: Option<u64>,
},
```

Using the `ics20-1` protocol is similar to using the `ibctransfer` module in Go

IBC transactions with CosmWasm

To connect two CosmWasm contracts over IBC you must establish an IBC channel between them. The IBC channel establishment process uses a four way handshake. Here is a summary of the steps:

1. **OpenInit** Declares to chain B the information used to verify the legitimacy of chain A, and ask for verification information
2. **OpenTry** After chain B verifies the legitimacy of chain A, it responds with its verification information.
3. **OpenAck** Once chain A verifies the legitimacy of chain B claims, is now ready to open the channel
4. **OpenConfirm** Once chain B receives the acknowledgement, it is ready to open che channel

IBC transactions with CosmWasm

You need to expose 6 entry points in a CosmWasm contract to enable IBC communication:

- **ibc_channel_open** to handle the **MsgChannelOpenInit** and **MsgChannelOpenTry** steps of the channel handshake.
- **ibc_channel_connect** to handle the **MsgChannelOpenAck** and **MsgChannelOpenConfirm** steps of the channel handshake.
- **ibc_channel_close** to handle channel closure.
- **ibc_packet_receive** to handle **MsgRecvPacket**.
- **ibc_packet_ack** to handle **MsgAcknowledgement**.
- **ibc_packet_timeout** to handle **MsgTimeout**.

IBC transactions with CosmWasm

You need to expose 6 entry points in a CosmWasm contract to enable IBC communication:

- **`ibc_channel_open`** to handle the **`MsgChannelOpenInit`** and **`MsgChannelOpenTry`** steps of the channel handshake.
- **`ibc_channel_connect`** to handle the **`MsgChannelOpenAck`** and **`MsgChannelOpenConfirm`** steps of the channel handshake.
- **`ibc_channel_close`** to handle channel closure.
- **`ibc_packet_receive`** to handle **`MsgRecvPacket`**.
- **`ibc_packet_ack`** to handle **`MsgAcknowledgement`**.
- **`ibc_packet_timeout`** to handle **`MsgTimeout`**.

Having implemented these methods, once you instantiate an instance of the contract it will be assigned a port. Ports identify a receiver on a blockchain in much the same way as ports identify applications on a computer.

CW20 ICS20

This IBC Enabled contract allows to send CW20 tokens from one chain over the standard ICS20 protocol to the bank module of another chain (send custom CW20 tokens with IBC and use them just like native tokens on other chains)

This particular contract above accepts cw20 tokens and sends those to a remote chain, as well as receiving the tokens back and releasing the original cw20 token to a new owner. It does not (yet) allow minting coins originating from the remote chain.

IBC Counter

When prompted on a chain, the IBC counter Smart Contract will:

- Send messages to its counterpart chain
- Count the number of times messages have been received on both sides.

At a high level, to use this contract:

1. Store and instantiate the contract on two IBC enabled chains (A and B).
2. The repository lets you configure and run a relayer to connect the two contracts. Anyway, as Archway currently run relayers for enabling channels for both Axelar and Osmosis, you can rely on those active relayers.
3. Execute the `Increment {}` method on one contract to increment the send a message and increment the count on the other one.
4. Use the `GetCount { connection }` query to determine the message count for a given connection.

https://github.com/0xekez/cw-ibc-example

IBC Counter

IBC allows us to send packets from chain A to chain B and get a responses.

The first step is to let the contract/module in chain A requesting to send a packet.

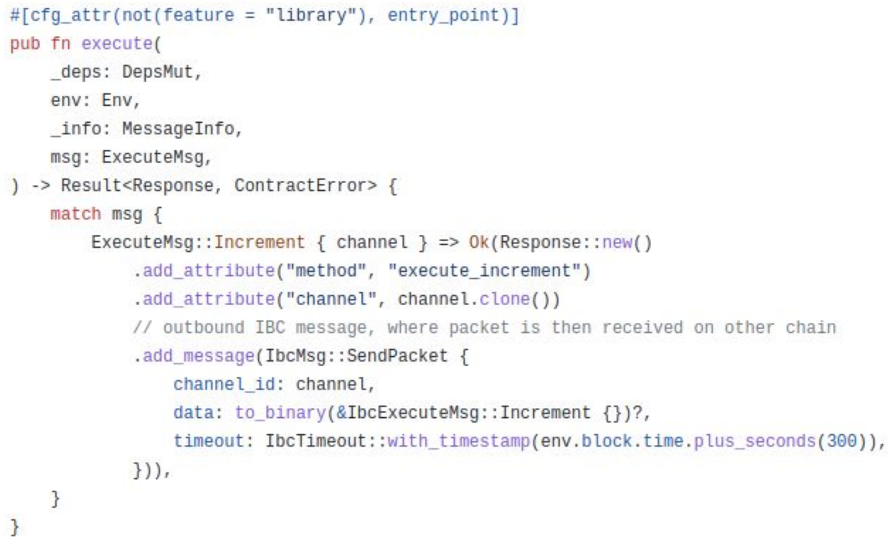
The packet is then relayed to chain B, where it "receives" the packet and calculates an "acknowledgement" (which may contain a success result or an error message. These are bytes that are then interpreted by the receiving contract).

The acknowledgement is then relayed back to chain A, completing the cycle.

As the packet may never be delivered before the timeout, calling the "timeout" handler on chain A abort the process.

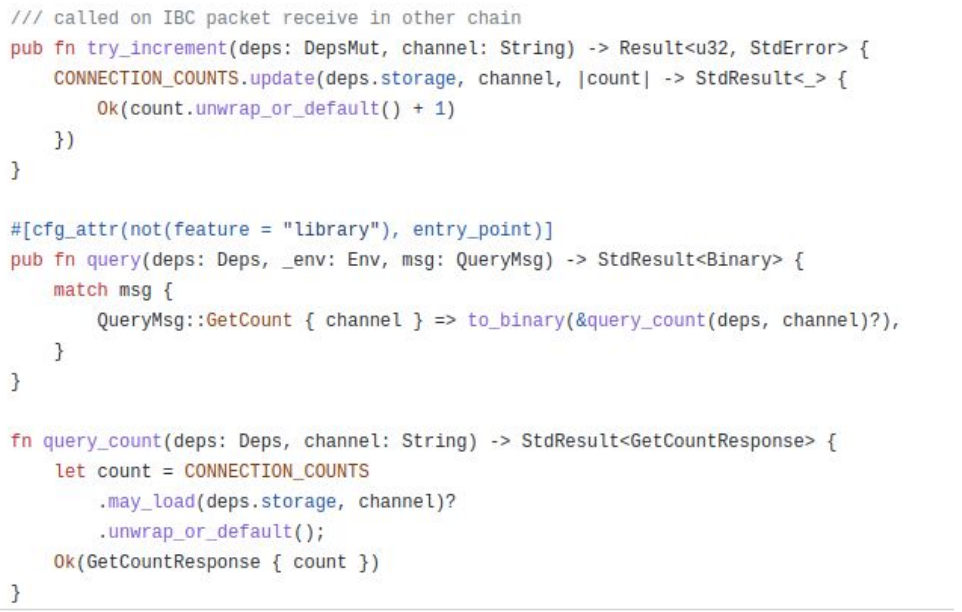
As in this case the chain A sends the packet and receive a "timeout" message. No "receive" nor "acknowledgement" callbacks are ever executed.

IBC Counter (contract.rs)



```rust
#[cfg_attr(not(feature = "library"), entry_point)]
pub fn execute(
    _deps: DepsMut,
    env: Env,
    _info: MessageInfo,
    msg: ExecuteMsg,
) -> Result<Response, ContractError> {
    match msg {
        ExecuteMsg::Increment { channel } => Ok(Response::new()
            .add_attribute("method", "execute_increment")
            .add_attribute("channel", channel.clone())
            // outbound IBC message, where packet is then received on other chain
            .add_message(IbcMsg::SendPacket {
                channel_id: channel,
                data: to_binary(&IbcExecuteMsg::Increment {})?,
                timeout: IbcTimeout::with_timestamp(env.block.time.plus_seconds(300)),
            })),
    }
}
```

When the `Increment` message for a particular channel is executed on a contract instance, a response is created that contains an `IbcMsg::SendPacket` message with the `Increment` message to execute on the counterparty.



```rust
/// called on IBC packet receive in other chain
pub fn try_increment(deps: DepsMut, channel: String) -> Result<u32, StdError> {
    CONNECTION_COUNTS.update(deps.storage, channel, |count| -> StdResult<_> {
        Ok(count.unwrap_or_default() + 1)
    })
}

#[cfg_attr(not(feature = "library"), entry_point)]
pub fn query(deps: Deps, _env: Env, msg: QueryMsg) -> StdResult<Binary> {
    match msg {
        QueryMsg::GetCount { channel } => to_binary(&query_count(deps, channel)?),
    }
}

fn query_count(deps: Deps, channel: String) -> StdResult<GetCountResponse> {
    let count = CONNECTION_COUNTS
        .may_load(deps.storage, channel)?
        .unwrap_or_default();
    Ok(GetCountResponse { count })
}
```

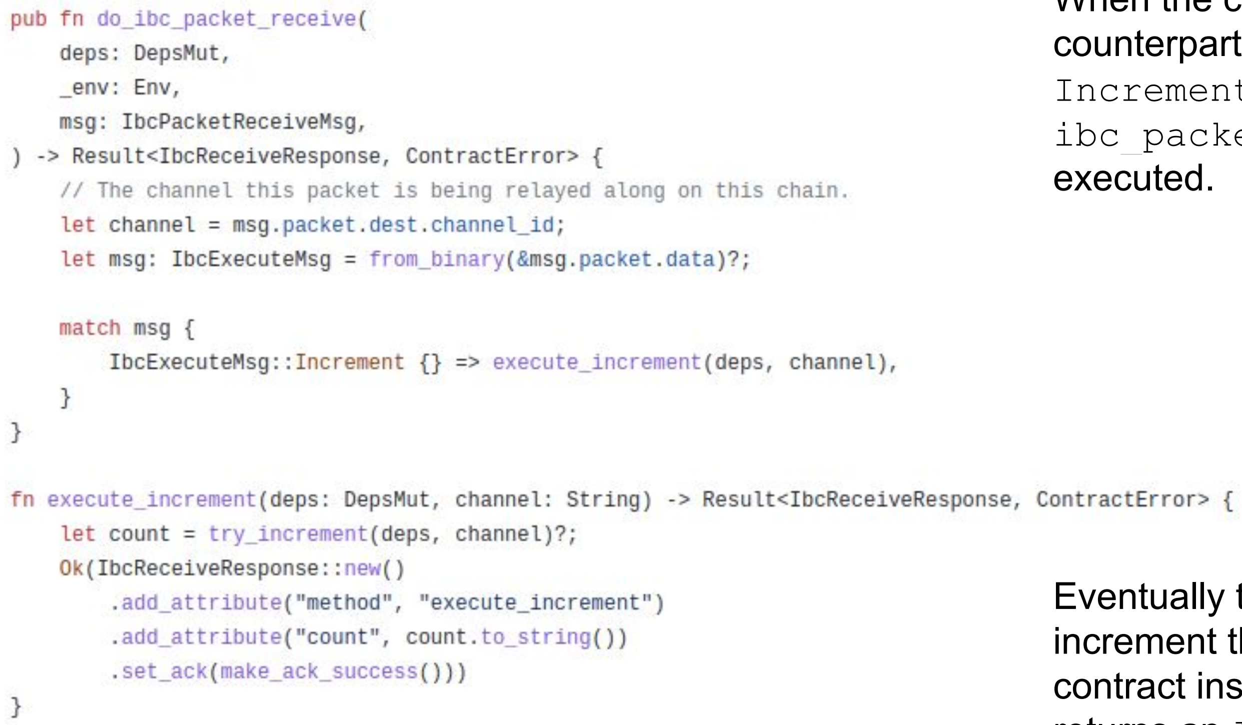
You can query the contract instance to retrieve the current value of the counter.

IBC Counter (ibc.rs)

```rust
/// Handles the `OpenInit` and `OpenTry` parts of the IBC handshake.
#[cfg_attr(not(feature = "library"), entry_point)]
pub fn ibc_channel_open(
    _deps: DepsMut,
    _env: Env,
    msg: IbcChannelOpenMsg,
) -> Result<IbcChannelOpenResponse, ContractError> {
    validate_order_and_version(msg.channel(), msg.counterparty_version())?;
    Ok(None)
}

#[cfg_attr(not(feature = "library"), entry_point)]
pub fn ibc_channel_connect(
    deps: DepsMut,
    _env: Env,
    msg: IbcChannelConnectMsg,
) -> Result<IbcBasicResponse, ContractError> {
    validate_order_and_version(msg.channel(), msg.counterparty_version())?;

    // Initialize the count for this channel to zero.
    let channel = msg.channel().endpoint.channel_id.clone();
    CONNECTION_COUNTS.save(deps.storage, channel.clone(), &0)?;

    Ok(IbcBasicResponse::new()
        .add_attribute("method", "ibc_channel_connect")
        .add_attribute("channel_id", channel))
}
```

```rust
#[cfg_attr(not(feature = "library"), entry_point)]
pub fn ibc_channel_close(
    deps: DepsMut,
    _env: Env,
    msg: IbcChannelCloseMsg,
) -> Result<IbcBasicResponse, ContractError> {
    let channel = msg.channel().endpoint.channel_id.clone();
    // Reset the state for the channel.
    CONNECTION_COUNTS.remove(deps.storage, channel.clone());
    Ok(IbcBasicResponse::new()
        .add_attribute("method", "ibc_channel_close")
        .add_attribute("channel", channel))
}

#[cfg_attr(not(feature = "library"), entry_point)]
pub fn ibc_packet_receive(
    deps: DepsMut,
    env: Env,
    msg: IbcPacketReceiveMsg,
) -> Result<IbcReceiveResponse, Never> {
    // Regardless of if our processing of this packet works we need to
    // commit an ACK to the chain. As such, we wrap all handling logic
    // in a seprate function and on error write out an error ack.
    match do_ibc_packet_receive(deps, env, msg) {
        Ok(response) => Ok(response),
        Err(error) => Ok(IbcReceiveResponse::new()
            .add_attribute("method", "ibc_packet_receive")
            .add_attribute("error", error.to_string())
            .set_ack(make_ack_fail(error.to_string()))),
    }
}
```

IBC Counter (ibc.rs)

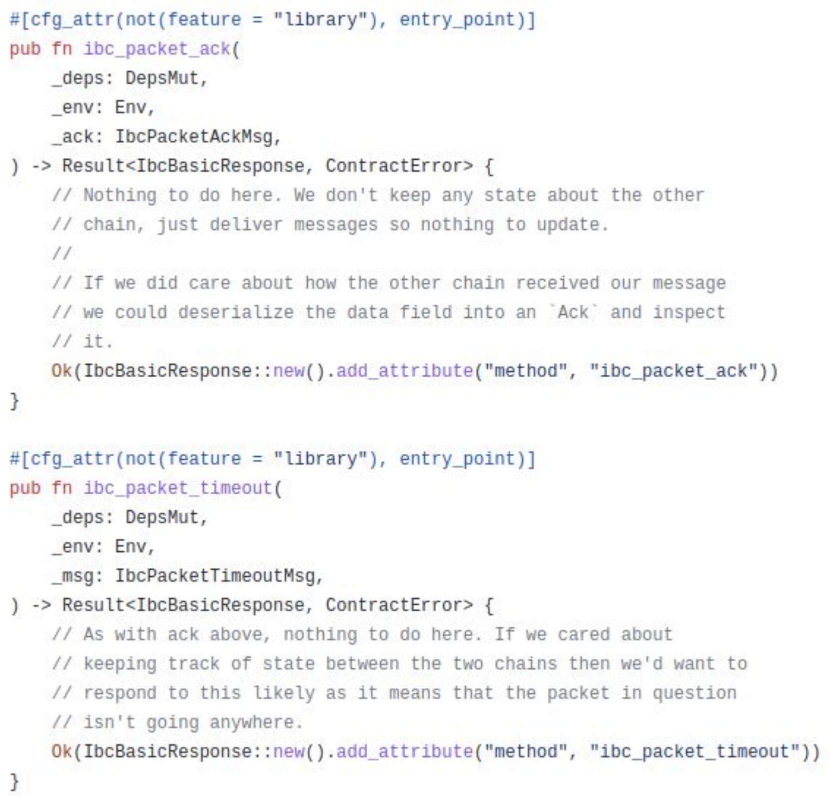


```rust
pub fn do_ibc_packet_receive(
    deps: DepsMut,
    _env: Env,
    msg: IbcPacketReceiveMsg,
) -> Result<IbcReceiveResponse, ContractError> {
    // The channel this packet is being relayed along on this chain.
    let channel = msg.packet.dest.channel_id;
    let msg: IbcExecuteMsg = from_binary(&msg.packet.data)?;

    match msg {
        IbcExecuteMsg::Increment {} => execute_increment(deps, channel),
    }
}

fn execute_increment(deps: DepsMut, channel: String) -> Result<IbcReceiveResponse, ContractError> {
    let count = try_increment(deps, channel)?;
    Ok(IbcReceiveResponse::new()
        .add_attribute("method", "execute_increment")
        .add_attribute("count", count.to_string())
        .set_ack(make_ack_success()))
}
```

When the contract instance on the counterparty chain receives the `Increment` message, the `ibc_packet_receive` function is executed.

Eventually the code tries to increment the counter in the contract instance's state and returns an `IbcReceiveResponse` that sets the acknowledgement.

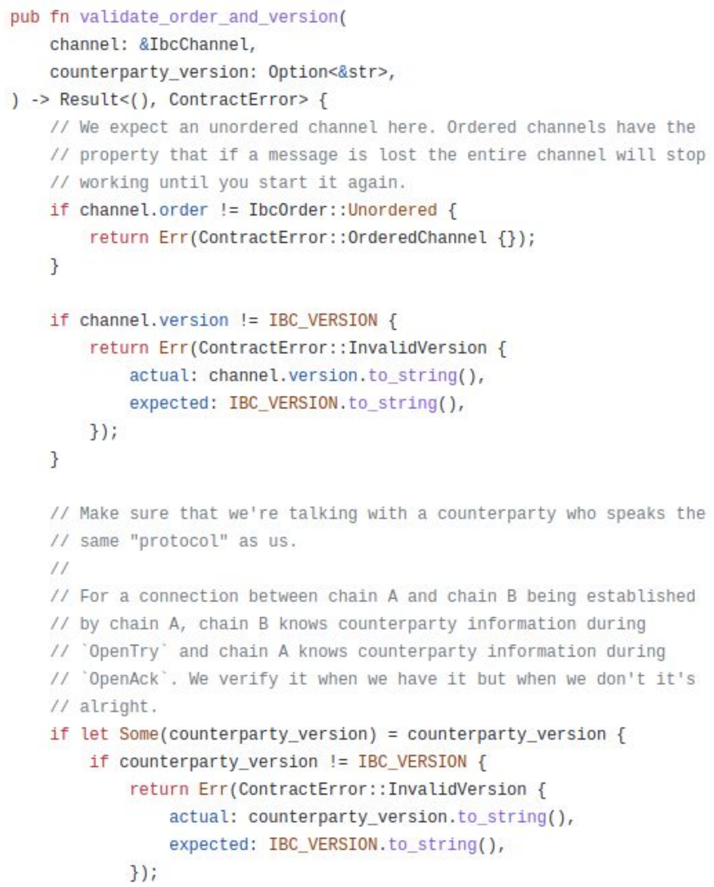
IBC Counter (ibc.rs)



```rust
#[cfg_attr(not(feature = "library"), entry_point)]
pub fn ibc_packet_ack(
    _deps: DepsMut,
    _env: Env,
    _ack: IbcPacketAckMsg,
) -> Result<IbcBasicResponse, ContractError> {
    // Nothing to do here. We don't keep any state about the other
    // chain, just deliver messages so nothing to update.
    //
    // If we did care about how the other chain received our message
    // we could deserialize the data field into an `Ack` and inspect
    // it.
    Ok(IbcBasicResponse::new().add_attribute("method", "ibc_packet_ack"))
}

#[cfg_attr(not(feature = "library"), entry_point)]
pub fn ibc_packet_timeout(
    _deps: DepsMut,
    _env: Env,
    _msg: IbcPacketTimeoutMsg,
) -> Result<IbcBasicResponse, ContractError> {
    // As with ack above, nothing to do here. If we cared about
    // keeping track of state between the two chains then we'd want to
    // respond to this likely as it means that the packet in question
    // isn't going anywhere.
    Ok(IbcBasicResponse::new().add_attribute("method", "ibc_packet_timeout"))
}
```

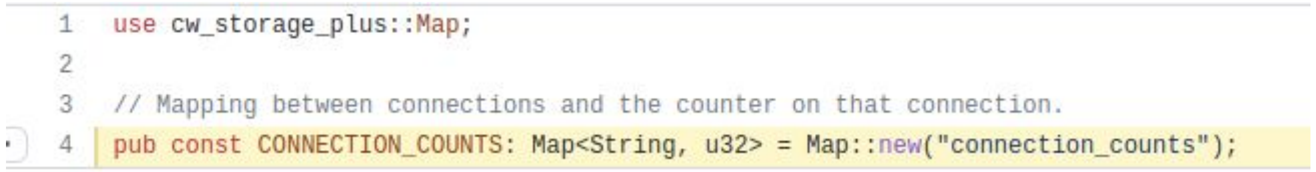
The processing of the acknowledgment happens in `ibc_packet_ack`.

Channel ordering and channel version validation is performed



```rust
pub fn validate_order_and_version(
    channel: &IbcChannel,
    counterparty_version: Option<&str>,
) -> Result<(), ContractError> {
    // We expect an unordered channel here. Ordered channels have the
    // property that if a message is lost the entire channel will stop
    // working until you start it again.
    if channel.order != IbcOrder::Unordered {
        return Err(ContractError::OrderedChannel {});
    }

    if channel.version != IBC_VERSION {
        return Err(ContractError::InvalidVersion {
            actual: channel.version.to_string(),
            expected: IBC_VERSION.to_string(),
        });
    }

    // Make sure that we're talking with a counterparty who speaks the
    // same "protocol" as us.
    //
    // For a connection between chain A and chain B being established
    // by chain A, chain B knows counterparty information during
    // `OpenTry` and chain A knows counterparty information during
    // `OpenAck`. We verify it when we have it but when we don't it's
    // alright.
    if let Some(counterparty_version) = counterparty_version {
        if counterparty_version != IBC_VERSION {
            return Err(ContractError::InvalidVersion {
                actual: counterparty_version.to_string(),
                expected: IBC_VERSION.to_string(),
            });
```

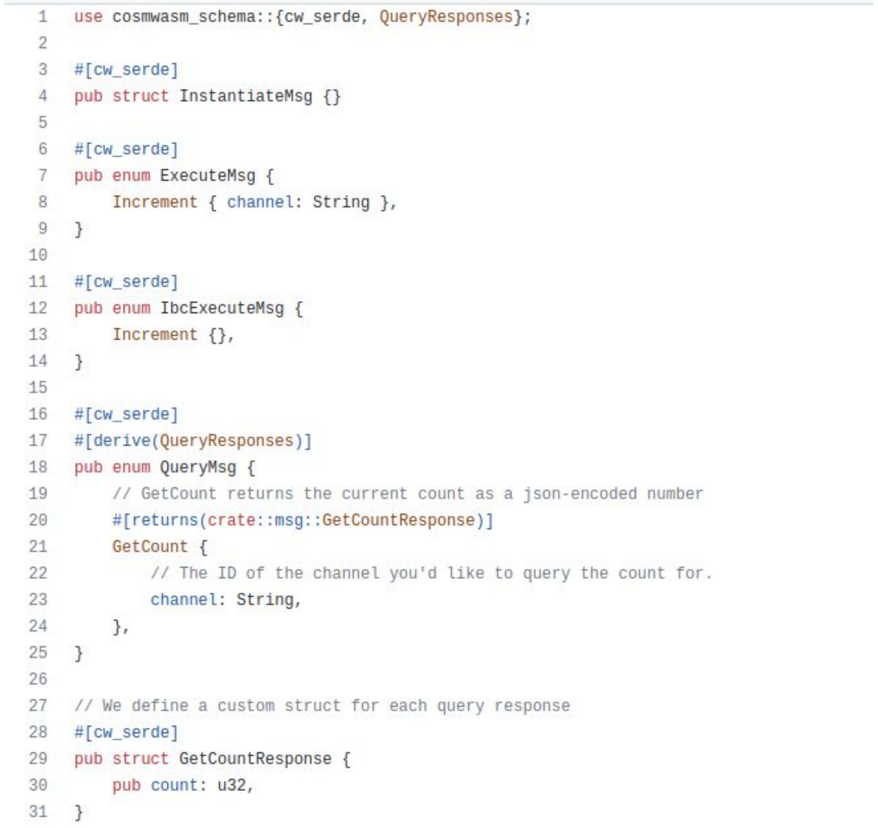
IBC Counter (state.rs)



```rust
use cw_storage_plus::Map;

// Mapping between connections and the counter on that connection.
pub const CONNECTION_COUNTS: Map<String, u32> = Map::new("connection_counts");
```

The state of the contract is simply a map with the channel ID as key and the counter for that channel as value.

IBC Counter (message.rs)



```rust
use cosmwasm_schema::{cw_serde, QueryResponses};

#[cw_serde]
pub struct InstantiateMsg {}

#[cw_serde]
pub enum ExecuteMsg {
    Increment { channel: String },
}

#[cw_serde]
pub enum IbcExecuteMsg {
    Increment {},
}

#[cw_serde]
#[derive(QueryResponses)]
pub enum QueryMsg {
    // GetCount returns the current count as a json-encoded number
    #[returns(crate::msg::GetCountResponse)]
    GetCount {
        // The ID of the channel you'd like to query the count for.
        channel: String,
    },
}

// We define a custom struct for each query response
#[cw_serde]
pub struct GetCountResponse {
    pub count: u32,
}
```

The messages to instantiate the contract, execute the increment message on the contract, send the increment message to the counterparty and query the contract

This contract only sends one type of packet data (i.e. the `Increment` message).

Resources

CosmWasm and IBC

https://github.com/cosmos/ibc-go/wiki/Cosmwasm-and-IBC

https://github.com/CosmWasm/cw-plus/tree/v0.6.0-beta1/contracts/cw20-ics20

https://github.com/CosmWasm/cosmwasm/blob/main/IBC.md

Relayers

https://github.com/archway-network/networks/tree/main/_IBC

https://tutorials.cosmos.network/hands-on-exercise/5-ibc-adv/3-go-relayer.html

https://github.com/cosmos/relayer

Resources

IBC

https://tutorials.cosmos.network/academy/3-ibc/1-what-is
-ibc.html#

CosmWasm

https://book.cosmwasm.com/

https://docs.archway.io/developers

https://area-52.io/

Examples:

https://github.com/CosmWasm/cosmwasm-plus/tree/v0.6.0-beta1/packages/cw20

https://github.com/CosmWasm/cw-plus/tree/v0.6.0-beta1/contracts/cw20-ics20

https://github.com/CosmWasm/cosmwasm/blob/main/IBC.md

Thanks!

Q&As