# #

gRPC Client

irishub v1.0.0 (depends on Cosmos-SDK v0.41) introduced Protobuf as the mainencodingopen in new windowlibrary, and this brings a wide range of Protobuf-based tools that can be plugged into the SDK. One such tool isgRPCopen in new window , a modern open source high performance RPC framework that has decent client support in several languages.

#

gRPC Server Port, Activation and Configuration

Thegrpc.Server is a concrete gRPC server, which spawns and serves any gRPC requests. This server can be configured inside~/.iris/config/app.toml :

- grpc.enable = true|false
- field defines if the gRPC server should be enabled. Defaults totrue
- .
- grpc.address = {string}
- field defines the address (really, the port, since the host should be kept at0.0.0.0
-) the server should bind to. Defaults to0.0.0.0:9000
- .

Once the gRPC server is started, you can send requests to it using a gRPC client.

#

gRPC Endpoints

An overview of all available gRPC endpoints shipped with the IRIShub isProtobuf documention .

#

Generating, Signing and Broadcasting Transactions

It is possible to manipulate transactions programmatically via Go using the Cosmos SDK'sTxBuilder interface.

#

Generating a Transaction

Before generating a transaction, a new instance of aTxBuilder needs to be created. Since the SDK supports both Amino and Protobuf transactions, the first step would be to decide which encoding scheme to use. All the subsequent steps remain unchanged, whether you're using Amino or Protobuf, asTxBuilder abstracts the encoding mechanisms. In the following snippet, we will use Protobuf.

import ("cosmossdk.io/simapp") func sendTx ( ) error { // Choose your codec: Amino or Protobuf. Here, we use Protobuf, given by the following function. encCfg:= simapp. MakeTestEncodingConfig ( ) // Create a new TxBuilder. txBuilder:= encCfg. TxConfig. NewTxBuilder ( ) // --snip-- } We can also set up some keys and addresses that will send and receive the transactions. Here, for the purpose of the tutorial, we will be using some dummy data to create keys.

import ("github.com/cosmos/cosmos-sdk/testutil/testdata") priv1, _ , addr1:= testdata. KeyTestPubAddr ( ) priv2, _ , addr2:= testdata. KeyTestPubAddr ( ) priv3, _ , addr3:= testdata. KeyTestPubAddr ( ) Populating theTxBuilder can be done via itsmethodsopen in new window :

import (banktypes"github.com/cosmos/cosmos-sdk/x/bank/types") func sendTx ( ) error { // --snip-- // Define two x/bank MsgSend messages: // - from addr1 to addr3, // - from addr2 to addr3. // This means that the transactions needs two signers: addr1 and addr2. msg1:= banktypes. NewMsgSend (addr1, addr3, types. NewCoins (types. NewInt64Coin ("atom" , 12))) msg2:= banktypes. NewMsgSend (addr2, addr3, types. NewCoins (types. NewInt64Coin ("atom" , 34))) err:= txBuilder. SetMsgs (msg1, msg2) if err!= nil { return err} txBuilder. SetGasLimit (...) txBuilder. SetFeeAmount (...) txBuilder. SetMemo (...) txBuilder. SetTimeoutHeight (...) } At this point,TxBuilder 's underlying transaction is ready to be signed.

#

Signing a Transaction

We set encoding config to use Protobuf, which will useSIGN_MODE_DIRECT by default. As perADR-020open in new window , each signer needs to sign theSignerInfo s of all other signers. This means that we need to perform two steps sequentially:

- for each signer, populate the signer'sSignerInfo
- insideTxBuilder
- ,
- once allSignerInfo
- s are populated, for each signer, sign theSignDoc
- (the payload to be signed).

In the currentTxBuilder 's API, both steps are done using the same method:SetSignatures() . The current API requires us to first perform a round ofSetSignatures() with empty signatures , only to populateSignerInfo s, and a second round ofSetSignatures() to actually sign the correct payload.

import (cryptotypes"github.com/cosmos/cosmos-sdk/crypto/types" "github.com/cosmos/cosmos-sdk/types/tx/signing" xauthsigning"github.com/cosmos/cosmos-sdk/x/auth/signing") func sendTx ( ) error { // --snip-- privs:= [ ] cryptotypes. PrivKey{ priv1, priv2} accNums:= [ ] uint64 { ... , ... } // The accounts' account numbers accSeqs:= [ ] uint64 { ... , ... } // The accounts' sequence numbers // First round: we gather all the signer infos. We use the "set empty // signature" hack to do that. var sigsV2[ ] signing. SignatureV2for i, priv:= range privs{ sigV2:= signing. SignatureV2{ PubKey: priv. PubKey ( ) , Data: & signing. SingleSignatureData{ SignMode: encCfg. TxConfig. SignModeHandler ( ) . DefaultMode ( ) , Signature: nil , } , Sequence: accSeqs[i] , } sigsV2= append (sigsV2, sigV2) } err:= txBuilder. SetSignatures (sigsV2...) if err!= nil { return err} // Second round: all signer infos are set, so each signer can sign. sigsV2= [ ] signing. SignatureV2{ } for i, priv:= range privs{ signerData:= xauthsigning. SignerData{ ChainID: chainID, AccountNumber: accNums[i] , Sequence: accSeqs[i] , } sigV2, err:= tx. SignWithPrivKey (encCfg. TxConfig. SignModeHandler ( ) . DefaultMode ( ) , signerData, txBuilder, priv, encCfg. TxConfig, accSeqs[i]) if err!= nil { return nil , err} sigsV2= append (sigsV2, sigV2) } err= txBuilder. SetSignatures (sigsV2...) if err!= nil { return err} } TheTxBuilder is now correctly populated. To print it, you can use theTxConfig interface from the initial encoding configencCfg :

func sendTx ( ) error { // --snip-- // Generated Protobuf-encoded bytes. txBytes, err:= encCfg. TxConfig. TxEncoder ( ) (txBuilder. GetTx ( )) if err!= nil { return err} // Generate a JSON string. txJSONBytes, err:= encCfg. TxConfig. TxJSONEncoder ( ) (txBuilder. GetTx ( )) if err!= nil { return err} txJSON:= string (txJSONBytes) }

#

Broadcasting a Transaction

The preferred way to broadcast a transaction is to use gRPC, though using REST (viagRPC-gateway) or the Tendermint RPC is also posible. For this tutorial, we will only describe the gRPC method.

import ("context" "fmt" "google.golang.org/grpc" "github.com/cosmos/cosmos-sdk/types/tx") func sendTx (ctx context. Context) error { // --snip-- // Create a connection to the gRPC server. grpcConn:= grpc. Dial ("127.0.0.1:9090" , // Or your gRPC server address. grpc. WithInsecure ( ) , // The SDK doesn't support any transport security mechanism.) defer grpcConn. Close ( ) // Broadcast the tx via gRPC. We create a new client for the Protobuf Tx // service. txClient:= tx. NewServiceClient (grpcConn) // We then call the BroadcastTx method on this client. grpcRes, err:= txClient. BroadcastTx (ctx, & tx. BroadcastTxRequest{ Mode: tx. BroadcastMode_BROADCAST_MODE_SYNC, TxBytes: txBytes, // Proto-binary of the signed transaction, see previous step. } ,) if err!= nil { return err} fmt. Println (grpcRes. TxResponse. Code) // Should be 0 if the tx is successful return nil }