Common Relayer Messaging for Crosschain Function Calls

Weijia Zhang, Peter Robinson, Aiman Baharna, Susumu Toriumi, Anais Ofranc, Chaals Nevile

IEEE ICBC Crosschain 2023

Abstract

The Crosschain Protocol Stack defines a way for enterprises to create interoperable components for crosschain communications.

Crosschain Applications use Crosschain Function calls to allow business logic to be executed across blockchains.

Components in the Crosschain Function Call layer rely on components in the Crosschain Messaging layer to deliver information from one blockchain to another such that the information can be trusted.

This specification defines the interface that components in the Crosschain Function Call layer can use to verify information from other blockchains.

Architecture

Crosschain Applications	Application code across blockchains
Crosschain Function Calls	Enables functions to be executed across blockchains
Crosschain Messaging	Enables events generated on one blockchain to be trusted on another blockchain

Scope

- Targets at Ethereum Virtual Machine (EVM) compatible blockchains.
- Defines interfaces using the Solidity programming language.
- Expects to expand to other programming languages

Relayer workflow

1. User application initiate a crosschain function call across one or more blockchains.

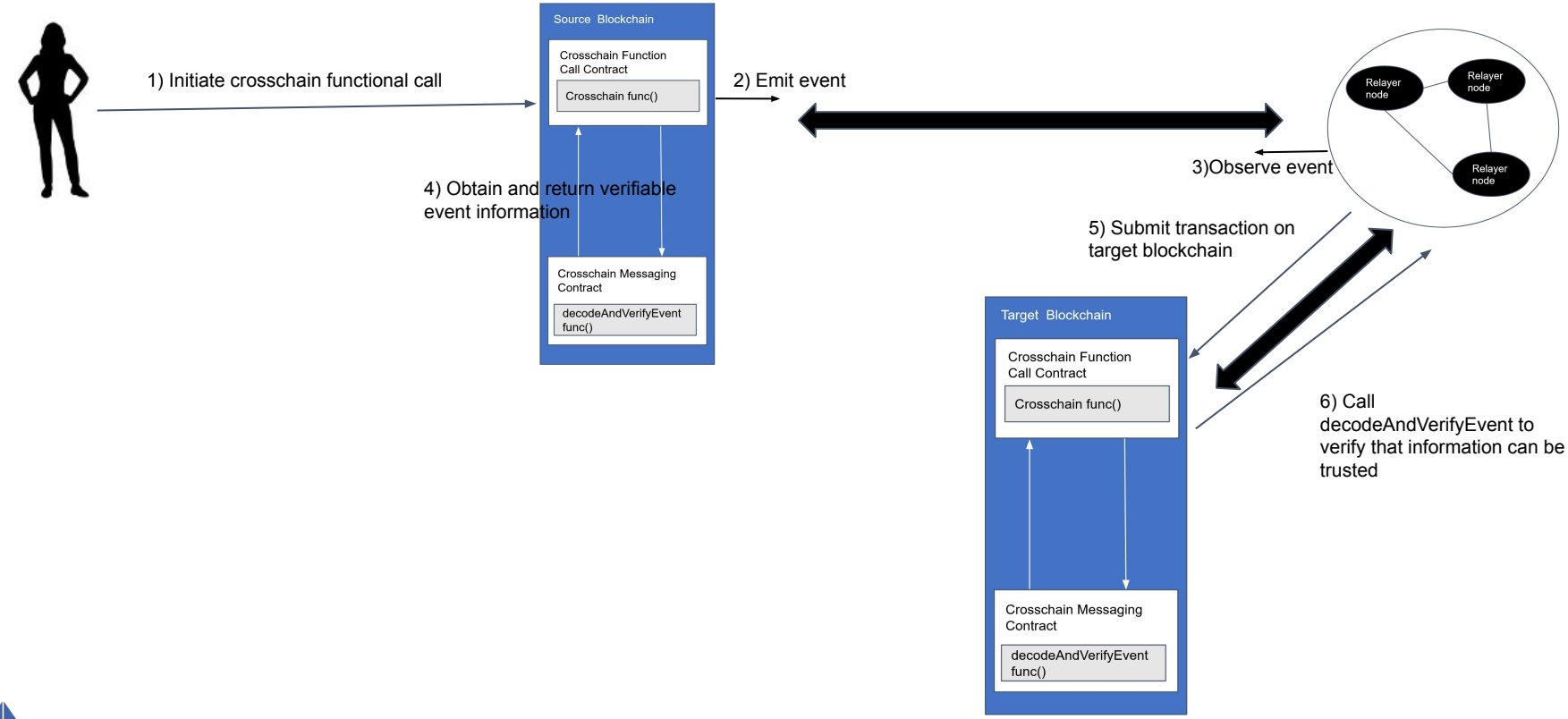
1. The Crosschain Function Call contract on the source blockchain emits an event.

1. Relayer nodes (also known as Attestors or Oracles, or Crosschain bridges) observe the blockchain for events being emitted by the Crosschain Function Call contract.

1. The Crosschain Function Call SDK code monitors for the event being emitted. The code calls the Crosschain Messaging Layer SDK code to ensure the Crosschain Messaging component knows which target blockchains the event will be needed on. The Crosschain Messaging SDK code creates or obtains verifiable event information and returns it to the Crosschain Function Call SDK code.

1. The Crosschain Function Call SDK code submits a transaction to the target blockchain, calling the Crosschain Function Call contract on the target blockchain. It supplies the event and signature or proof information from the Crosschain Messaging Layer SDK code. The Crosschain Function Call contract calls the Crosschain Messaging contract for the source blockchain calling decodeAndVerifyEvent. This call verifies that the event information did come from the source blockchain and can be trusted.

Relayer workflow (visual)



Protocol workflow

1. User application initiate a crosschain function call across one or more blockchains.

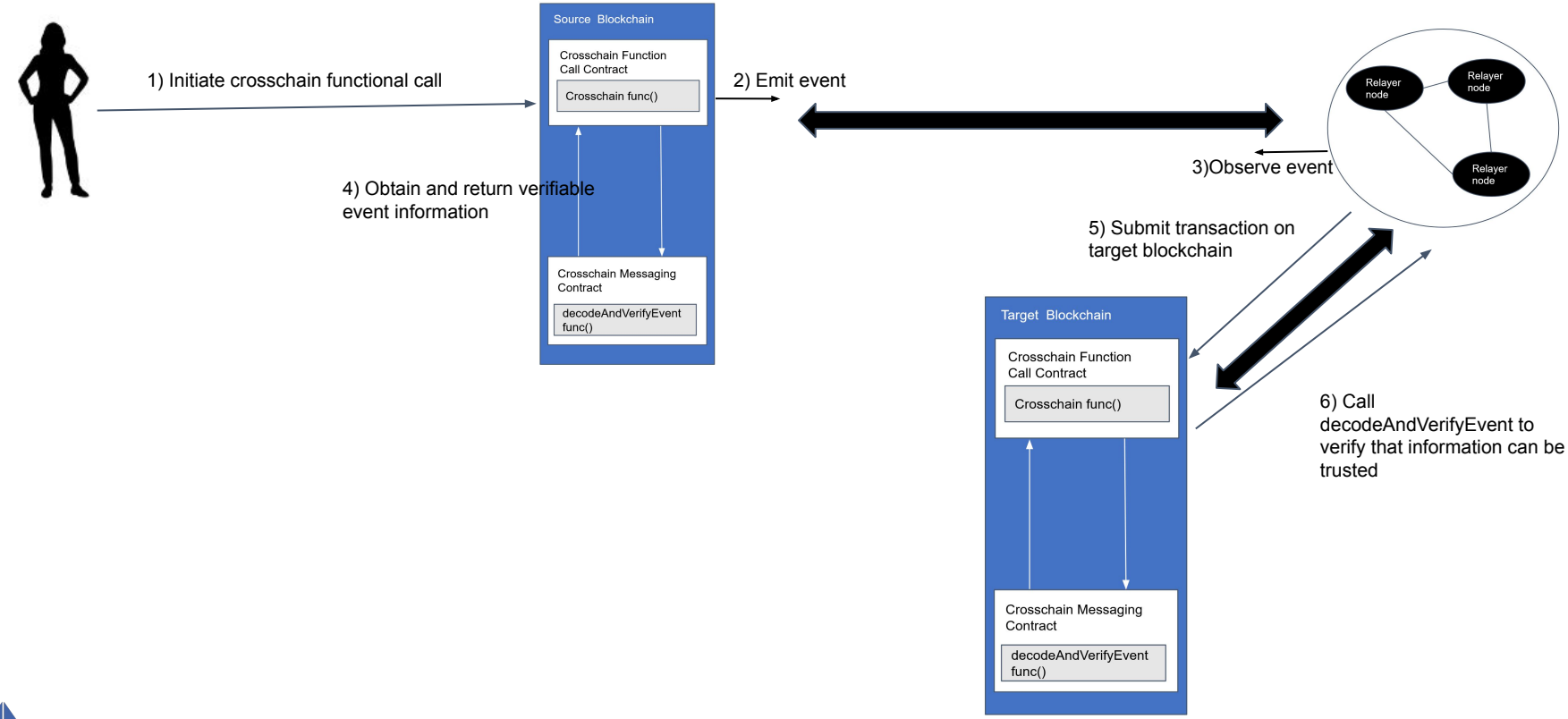
1. The Crosschain Function Call contract on the source blockchain emits an event.

1. Relayer nodes (also known as Attestors or Oracles, or Crosschain bridges) observe the blockchain for events being emitted by the Crosschain Function Call contract.

1. The Crosschain Function Call SDK code monitors for the event being emitted. The code calls the Crosschain Messaging Layer SDK code to ensure the Crosschain Messaging component knows which target blockchains the event will be needed on. The Crosschain Messaging SDK code creates or obtains verifiable event information and returns it to the Crosschain Function Call SDK code.

1. The Crosschain Function Call SDK code submits a transaction to the target blockchain, calling the Crosschain Function Call contract on the target blockchain. It supplies the event and signature or proof information from the Crosschain Messaging Layer SDK code. The Crosschain Function Call contract calls the Crosschain Messaging contract for the source blockchain calling decodeAndVerifyEvent. This call verifies that the event information did come from the source blockchain and can be trusted.

Protocol workflow (visual)



Interfaces

Solidity Interfaces

```
pragma solidity >=0.8;
interface CrosschainVerifier {
  function decodeAndVerifyEvent(uint256 _blockchainId, bytes32 _eventSig,
    bytes calldata _encodedInfo, bytes calldata _signatureOrProof)
    external view;
}
```

Interface parameters

- _blockchainId: The blockchain that emitted the event. This could be used to determine which sets of signing keys should be used to verify the signature parameter. The _blockchainId must be in EIP 3220 format.

- _eventSig: The event function-signature hash. This value is emitted as part of an event. It identifies which event was emitted.

- _encodedInfo: The abi.encodePacked of the blockchain identifier (_blockchainId), the Crosschain Control
contract's address, the event function signature (_eventSig), and the event data.

- _signatureOrProof: Signatures or proof information that an implementation can use to check that

  _encodedInfo is valid.

Signature formats

```
pragma solidity >=0.8;
struct Signature {
  uint256 by;
  uint256 sigR;
  uint256 sigS;
  uint8   sigV;
  bytes   meta;
}
struct Signatures {
  uint256 typ;
  Signature[] signatures;
}
```

Signature parameters

- by: The 160-bit Ethereum address derived from the 257-bit ECDSA public key of the signer.
- sigR: The ECDSA signature's R value.
- sigS: The ECDSA signature's S value.
- sigV: The ECDSA signature's V value.
- meta: The ECDSA signature's metadata can contain optional information on the platform (e.g. Ethereum), curve (e.g. SECP256K1) and hashing function (e.g. KECCAK-256) used to create the signature.
- typ: The type of signature or proof. For multiple ECDSA signatures, this is always 0x0001.
- signatures: An array of signatures. The length of the array must match numberOfSignatures.

Implementation creating relaying messages

- Getting blockchain ids
- Get event message information
- Getting signatures
- Package transactions
- Call standardized interface

Implementation creating relaying messages

- //Step 1. Get eventSig
  var eventSig = web3_utils.solidity Sha3(eventFunc);

- //Step 2. Get encodeInfo
  var encodedInfo = web3_utils.encodePacked(""+blockchainId+"", ""+crosschainControlAddr+"",
  eventSig, eventData);

Implementation creating relaying messages

- //Step 3 Sign EncodedInfo

```
for (var i=0; i<keys_priv.length; i++) {
var key = ec.keyFromPrivate(keys_priv[i]);

//calculate v from chain_id

var chainid_bn = BigInt(crosschainControlAddr);

var v = BigInt(y_parity)+chainid_bn*BigInt(2)+BigInt(35);

var signa_msg = key.sign(encodedInfo);
var signa = new Signature(crosschainControlAddr, signa_msg.r.toString(), signa_msg.s.toString(),
v.toString(), 'secp256k1');
signatures.push(signa);
}
```

Implementation creating relaying messages

- Step 4 Package Signatures

  var Signa_record = new Signatures(typ, signatures);

  var signatureOrProof = web3_utils.encodePacked(JSON.stringify(Signa_record));

- Step 5 Package Call Crosschaint functions and call Message Relayer Interface

Pending issues and considerations

- Discrepancy of different implementation of relaying messages
- Finality consideration
- Relayer Public key verification

Future Work

- Reference implementation of Relayer specification
- Working framework of crosschain function call framework (Define CroshchainExecutor Interface)

Thank you !

Any questions