L1-L2 Messaging
A crucial feature of a Layer 2 is its ability to interact with Layer 1.
Starknet has its own `L1-L2` messaging system, which is different from its consensus mechanism and the submission of state updates on L1. Messaging is a way for smart-contracts on L1 to interact with smart-contracts on L2 (or the other way around), allowing us to do "cross-chain" transactions. For example, we can do some computations on a chain and use the result of this computation on the other chain.
Bridges on Starknet all use `L1-L2` messaging. Let's say that you want to bridge tokens from Ethereum to Starknet. You will simply have to deposit your tokens in the L1 bridge contract, which will automatically trigger the minting of the same token on L2. Another good use case for `L1-L2` messaging would be [DeFi pooling][defi pooling doc].
On Starknet, it's important to note that the messaging system is **asynchronous** and **asymmetric**.
- **Asynchronous**: this means that in your contract code (being Solidity or Cairo), you can't wait the result of the message being sent on the other chain within your contract code execution.
- **Asymmetric**: sending a message from Ethereum to Starknet (`L1->L2`) is fully automated by the Starknet sequencer, which means that the message is being automatically delivered to the target contract on L2. However, when sending a message from Starknet to Ethereum (`L2->L1`), only the hash of the message is sent on L1 by the Starknet sequencer. You must then consume the message manually via a transaction on L1.
Let's dive into the details.
[defi pooling doc]: https://starkware.co/resource/defi-pooling/
The StarknetMessaging Contract
The crucial component of the `L1-L2` Messaging system is the [`StarknetCore`] [starknetcore etherscan] contract. It is a set of Solidity contracts deployed on Ethereum that allows Starknet to function properly. One of the contracts of `StarknetCore` is called `StarknetMessaging` and it is the contract responsible for passing messages between Starknet and Ethereum. `StarknetMessaging` follows an [interface] [IStarknetMessaging] with functions allowing to send message to L2, receiving messages on L1 from L2 and canceling messages.
```js
interface IStarknetMessaging is IStarknetMessagingEvents {
    function sendMessageToL2(
        uint256 toAddress,
        uint256 selector,
        uint256[] calldata payload
    ) external returns (bytes32);
    function consumeMessageFromL2(uint256 fromAddress, uint256[] calldata payload)
        external
        returns (bytes32);
    function startL1ToL2MessageCancellation(
        uint256 toAddress,
        uint256 selector,
        uint256[] calldata payload,
```

```
        uint256 nonce
    ) external;
    function cancelL1ToL2Message(
        uint256 toAddress,
        uint256 selector,
        uint256[] calldata payload,
        uint256 nonce
    ) external;
}
```

 Starknet messaging contract interface

In the case of `L1->L2` messages, the Starknet sequencer is constantly listening to the logs emitted by the `StarknetMessaging` contract on Ethereum.

Once a message is detected in a log, the sequencer prepares and executes a `L1HandlerTransaction` to call the function on the target L2 contract. This takes up to 1-2 minutes to be done (few seconds for ethereum block to be mined, and then the sequencer must build and execute the transaction).

`L2->L1` messages are prepared by contracts execution on L2 and are part of the block produced. When the sequencer produces a block, it sends the hash of each message prepared by contracts execution
to the `StarknetCore` contract on L1, where they can then be consumed once the block they belong to is proven and verified on Ethereum (which for now is around 3-4 hours).

[starknetcore etherscan]: https://etherscan.io/address/0xc662c410C0ECf747543f5bA90660f6ABeBD9C8c4

[IStarknetMessaging]: https://github.com/starkware-libs/cairo-lang/blob/4e233516f52477ad158bc81a86ec2760471c1b65/src/starkware/starknet/eth/IStarknetMessaging.sol#L6

Sending Messages from Ethereum to Starknet

If you want to send messages from Ethereum to Starknet, your Solidity contracts must call the `sendMessageToL2` function of the `StarknetMessaging` contract. To receive these messages on Starknet, you will need to annotate functions that can be called from L1 with the `#[l1_handler]` attribute.

Let's take a simple contract taken from [this tutorial][messaging contract] where we want to send a message to Starknet.

The `_snMessaging` is a state variable already initialized with the address of the `StarknetMessaging` contract. You can check all Starknet contract and sequencer addresses [here][starknet addresses].

```js
// Sends a message on Starknet with a single felt.
function sendMessageFelt(
    uint256 contractAddress,
    uint256 selector,
    uint256 myFelt
)
    external
    payable
```

```
{
    // We "serialize" here the felt into a payload, which is an array of uint256.
    uint256[] memory payload = new uint256[](1);
    payload[0] = myFelt;
    // msg.value must always be >= 20_000 wei.
    _snMessaging.sendMessageToL2{value: msg.value}(
        contractAddress,
        selector,
        payload
    );
}
```

The function sends a message with a single felt value to the `StarknetMessaging` contract.
Please note that if you want to send more complex data you can. Just be aware that your Cairo contract will only understand `felt252` data type. So you must ensure that the serialization of your data into the `uint256` array follow the Cairo serialization scheme.
It's important to note that we have `{value: msg.value}`. In fact, the minimum value we've to send here is `20k wei`, due to the fact that the `StarknetMessaging` contract will register
the hash of our message in the storage of Ethereum.
In addition to those `20k wei`, since the `L1HandlerTransaction` executed by the sequencer is not tied to any account (the message originates from L1), you must also ensure
that you pay enough fees on L1 for your message to be deserialized and processed on L2.
The fees of the `L1HandlerTransaction` are computed in a regular manner as it would be done for an `Invoke` transaction. For this, you can profile
the gas consumption using `starkli` or `snforge` to estimate the cost of your message execution.
The signature of the `sendMessageToL2` is:

```js
function sendMessageToL2(
        uint256 toAddress,
        uint256 selector,
        uint256[] calldata payload
    ) external override returns (bytes32);
```

The parameters are as follows:
- `toAddress`: The contract address on L2 that will be called.
- `selector`: The selector of the function of this contract at `toAddress`. This selector (function) must have the `#[l1_handler]` attribute to be callable.
- `payload`: The payload is always an array of `felt252` (which are represented by `uint256` in Solidity). For this reason we've inserted the input `myFelt` into the array.
  This is why we need to insert the input data into an array.

On the Starknet side, to receive this message, we have:
```cairo,noplayground
{{#include ../listings/ch16-building-advanced-starknet-smart-contracts/
no_listing_03_L1_L2_messaging/src/lib.cairo:felt_msg_handler}}
```
We need to add the `#[l1_handler]` attribute to our function. L1 handlers are special functions that can only be executed by a `L1HandlerTransaction`. There is nothing particular to do to receive transactions from L1, as the message is relayed by the sequencer automatically. In your `#[l1_handler]` functions, it is important to verify the sender of the L1 message to ensure that our contract can only receive messages from a trusted L1 contract.
[messaging contract]: https://github.com/glihm/starknet-messaging-dev/blob/main/solidity/src/ContractMsg.sol
[starknet addresses]: https://docs.starknet.io/documentation/tools/important_addresses/
Sending Messages from Starknet to Ethereum
When sending messages from Starknet to Ethereum, you will have to use the `send_message_to_l1` syscall in your Cairo contracts. This syscall allows you to send messages to the `StarknetMessaging` contract on L1. Unlike `L1->L2` messages, `L2->L1` messages must be consumed manually, which means that you will need your Solidity contract to call the `consumeMessageFromL2` function of the `StarknetMessaging` contract explicitly in order to consume the message.
To send a message from L2 to L1, what we would do on Starknet is:
```cairo,noplayground
{{#include ../listings/ch16-building-advanced-starknet-smart-contracts/
no_listing_03_L1_L2_messaging/src/lib.cairo:felt_msg_send}}
```
We simply build the payload and pass it, along with the L1 contract address, to the syscall function.
On L1, the important part is to build the same payload sent by the L2. Then you call `consumeMessageFromL2` in you Solidity contract by passing the L2 contract address and the payload. Please be aware that the L2 contract address expected by the `consumeMessageFromL2` is the address of the contract that sends the message on the L2 by calling `send_message_to_l1_syscall`.
```js
function consumeMessageFelt(
    uint256 fromAddress,
    uint256[] calldata payload
)
    external
{
    let messageHash = _snMessaging.consumeMessageFromL2(fromAddress, payload);
    // You can use the message hash if you want here.
    // We expect the payload to contain only a felt252 value (which is a uint256 in
Solidity).
    require(payload.length == 1, "Invalid payload");
    uint256 my_felt = payload[0];
```

```
    // From here, you can safely use `my_felt` as the message has been verified by StarknetMessaging.
    require(my_felt > 0, "Invalid value");
}
```

As you can see, in this context we don't have to verify which contract from L2 is sending the message (as we do on the L2 to verify which contract from L1 is sending the message). But we are actually using the `consumeMessageFromL2` of the `StarknetCore` contract to validate the inputs (the contract address on L2 and the payload) to ensure we are only consuming valid messages.

> **Note:** The `consumeMessageFromL2` function of the `StarknetCore` contract is expected to be called from a Solidity contract, and not directly on the `StarknetCore` contract. The reason of that is because the `StarknetCore` contract is using `msg.sender` to actually compute the hash of the message. And this `msg.sender` must correspond to the `to_address` field that is given to the function `send_message_to_l1_syscall` that is called on Starknet.

It is important to remember that on L1 we are sending a payload of `uint256`, but the basic data type on Starknet is `felt252`; however, `felt252` are approximately 4 bits smaller than `uint256`. So we have to pay attention to the values contained in the payload of the messages we are sending. If, on L1, we build a message with values above the maximum `felt252`, the message will be stuck and never consumed on L2.

Cairo Serde

Before sending messages between L1 and L2, you must remember that Starknet contracts, written in Cairo, can only understand serialized data. And serialized data is always an array of `felt252`.

In Solidity we have `uint256` type, and `felt252` are approximately 4 bits smaller than `uint256`. So we have to pay attention to the values contained in the payload of the messages we are sending.

If, on L1, we build a message with values above the maximum `felt252`, the message will be stuck and never consumed on L2.

So for instance, an actual `uint256` value in Cairo is represented by a struct like:

```cairo,does_not_compile
struct u256 {
    low: u128,
    high: u128,
}
```

which will be serialized as **TWO** felts, one for the `low`, and one for the `high`. This means that to send only one `u256` to Cairo, you'll need to send a payload from L1 with **TWO** values.

```js
uint256[] memory payload = new uint256[](2);
// Let's send the value 1 as a u256 in cairo: low = 1, high = 0.
payload[0] = 1;
payload[1] = 0;
```

If you want to learn more about the messaging mechanism, you can visit the [Starknet documentation][starknet messaging doc].
You can also find a [detailed guide here][glihm messaging guide] to test the messaging system locally.
[starknet messaging doc]: https://docs.starknet.io/documentation/architecture_and_concepts/Network_Architecture/messaging-mechanism/
[glihm messaging guide]: https://github.com/glihm/starknet-messaging-dev