Starknet / Cairo continued

Events in Starknet Contracts

See docs

The mechanism is similar to that on Ethereum.

A contract may emit events throughout its execution. Each event contains the following fields:

- `from_address`: address of the contract emitting the events
- `keys`: a list of field elements
- `data`: a list of field elements

The keys can be used for indexing the events, while the data may contain any information that we wish to log (note that we are dealing with two separate lists of possibly varying size, rather than a list of key-value pairs).

Events can be defined in a contract using the `@event` decorator. Once an event `E` has been defined, the compiler automatically adds the function `E.emit()`. The following example illustrates how an event is defined and emitted:

```
@event
func message_received(a : felt, b: felt):end
```

...

```
message_received.emit(1, 2);
```

The emit function emits an event with a single key, which is an identifier of the event, given by `sn_keccak(event_name)`

Useful Cairo equivalents for Solidity patterns

See Repo

For example

```
struct exampleStruct:
        member first_member : felt
        member second_member : felt
        member third_member : felt

end

# Cairo doesn't have enums, but you can abuse a struct as an enum. For
example:

struct exampleEnum:

        member first_process: felt
        member second_process: felt
        member third_process: felt

end

#### Setters ####

exampleStruct.first_member = 1
exampleStruct.second_member = 2
exampleStruct.third_member = 3

# to as an array

@storage_var
func _structs(struct_id: felt) -> (res : exampleStruct)
end

let new_struct = exampleStruct(
first_member=1
second_member=2
third_member=3
)
```

```
_structs.write(struct_id,new_struct)
```

Transaction Process

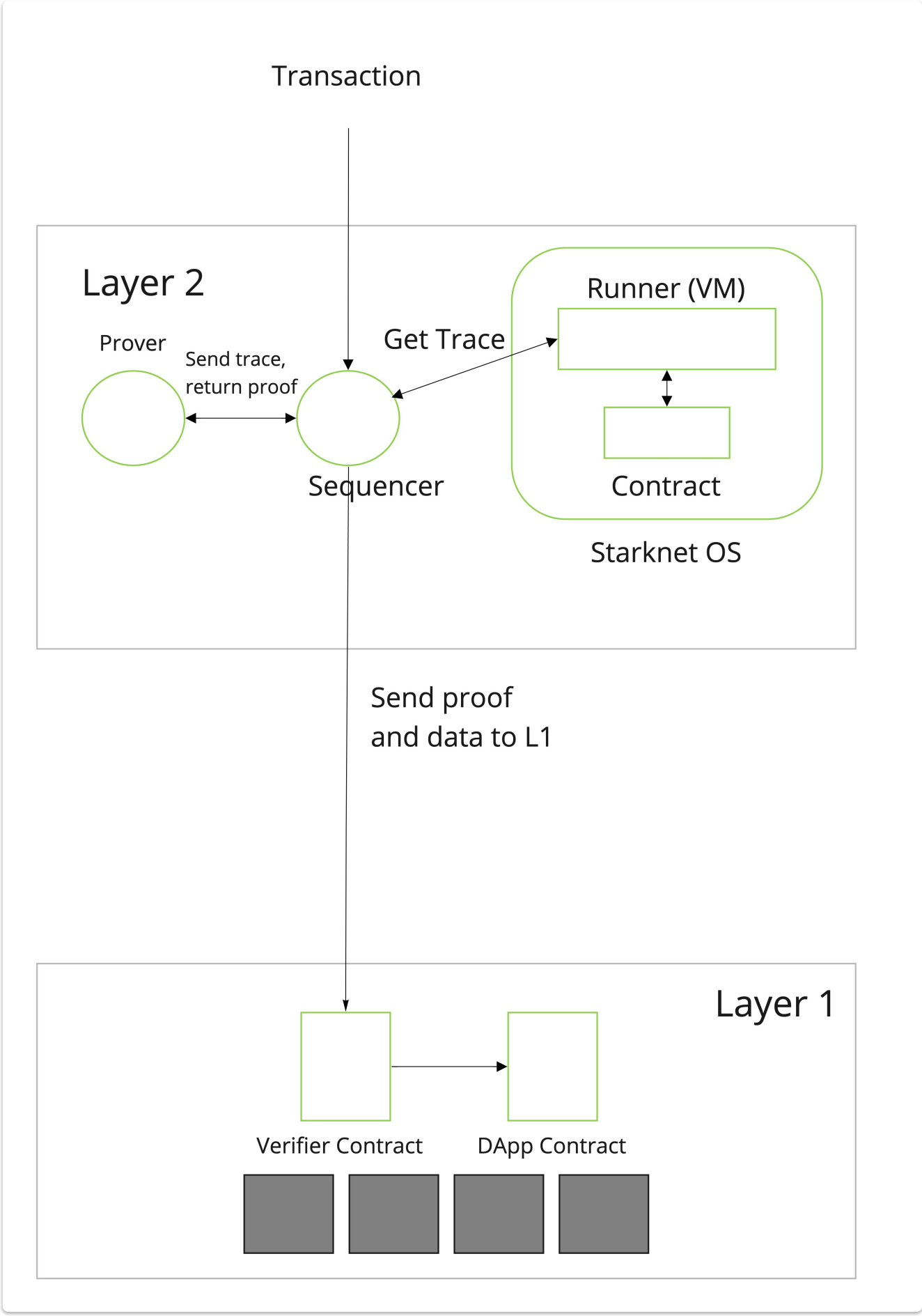
Accepts input from the client

Cairo contract creates state changes

Produces a list of changes to storage / balances.

Creates a proof of correct computation of all the transactions

Verifier contract checks the validity of the proof and applies the state changes.



Starknet has blocks, see Block structure
which consists of a header and a set of transactions
For further details see starknet documentation

Transaction Status

1. NOT_RECEIVED

Transaction is not yet known to the sequencer

2. RECEIVED

Transaction was received by the sequencer. Transaction will now either execute successfully or be rejected.

3. PENDING

Transaction executed successfully and entered the pending block.

4. REJECTED

Transaction executed unsuccessfully and thus was skipped (applies both to a pending and an actual created block). Possible reasons for transaction rejection:

- An assertion failed during the execution of the transaction (in StarkNet, unlike in Ethereum, transaction executions do not always succeed).
- The block may be rejected on L1, thus changing the transaction status to `REJECTED`

5. ACCEPTED_ON_L2

Transaction passed validation and entered an actual created block on L2.

6. ACCEPTED_ON_L1
   Transaction was accepted on-chain.

Transaction types

1. Deploy
2. Invoke
3. Declare

Invoke Transaction Structure

Name	Type	Description
`contract_address`	`FieldElement`	The address of the contract invoked by this transaction
`entry_point_selector`	`FieldElement`	The encoding of the selector for the function invoked (the entry point in the contract)
`calldata`	`List<FieldElement>`	The arguments passed to the invoked function
`signature`	`List<FieldElement>`	Additional information given by the caller, representing the signature of the transaction
`max_fee`	`FieldElement`	The maximum fee that the sender is willing to pay for the transaction
`version`	`FieldElement`	The transaction's version [1]

Data Availability

Currently Starknet is operating in zkrollup mode, this means that upon the acceptance of a state update on-chain, the state diff between the previous and new state is sent as calldata to Ethereum.

This data allows anyone that observes Ethereum to reconstruct the current state of StarkNet.
Note that to update the StarkNet state on L1, it suffices to send a valid proof – without information on the transactions or particular changes that this update caused.

There are other modes possible :

In ZK-Rollup mode data is published on-chain.
In Validium mode data is stored off-chain.
Volition is a hybrid data availability mode, where the user can choose whether to place data on-chain or off-chain.

Starknet Fee Mechanism

See docs

Users can specify the maximum fee that they are willing to pay for a transaction via the `max_fee` field.

The only limitation on the sequencer (enforced by the StarkNet OS) is that the actual fee charged is bounded by `max_fee`, but for now, StarkWare's sequencer will only charge the fee required to cover the proof cost (potentially less than the max fee).

Presently, the sequencer only takes into account L1 costs involving proof submission. There are two components affecting the L1 footprint of a transaction:

- computational complexity: the heavier the transaction, the larger its portion in the proof verification cost.
- on chain data: L1 calldata cost originating from data availability and L2→L1 messages.

The fee is charged atomically with the transaction execution on L2. The StarkNet OS injects a transfer of the fee-related ERC-20, with an amount equal to the fee paid, sender equals to the transaction submitter, and the sequencer as a receiver.

Reverting transactions and a problem

See documention for adding error messages
For example

```
with_attr error_message("ERC20: decimals exceed 2^8"):
assert_lt(decimals, UINT8_MAX)
end
```

But handling reverted transactions is being debated

Cairo is not natively capable of proving reverted transactions. The reason is that for a transaction to fail, one needs to prove that the transaction resulted in an error.
On the current version of StarkNet Alpha, either a transaction is valid and can be included in a proof or it is unprovable and cannot be included.
The problem this triggers is that an unprovable transaction cannot pay fees and one can imagine sending expensive computation-wise transactions which consume sequencers' resources without compensating for them. This allows DoS attacks on the sequencers.

Suggested solutions

1. Making All Transactions Provable
   This can be achieved by using a new programming language (whose syntax can be very similar to Cairo), in which the compiled contract will include the code to handle exceptions.
2. Economic Solution - "Red/Green"
   The main idea is that a sequencer can choose to include a transaction without executing it and receive a red fee, thus enabling it to ensure it receives a fee payment for their work. We can design the mechanism in such a way that a Sequencer will always be incentivized to execute a provable transaction and take the green fee instead of the red fee.
3. Transaction level PoW
   Users provide a PoW on top of their transactions based on the numbers of reverted txs. In such a way, a user would have to provide a small proof of work or none on the average case, but in the case too many reversions, the PoW will be much larger.

Useful Libraries

Import the libraries using this format

```
from starkware.cairo.common.bitwise import bitwise_operations
```

Math.cairo

- `assert_not_zero()`.
- `assert_not_equal()`.
- `assert_nn()`.
- `assert_le()`.
- `assert_lt()`.
- `assert_nn_le()`.
- `assert_in_range()`.
- `assert_le_250_bit()`.
- `split_felt()`.
- `assert_le_felt()`.
- `abs_value()`.
- `sign()`.
- `unsigned_div_rem()`.
- `signed_div_rem()`.

Signature.cairo

Common Library

- alloc.
- bitwise.
- cairo_builtins.
    - This has structs
        - BitwiseBuiltin
        - HashBuiltin
        - SignatureBuiltin
- default_dict.
- dict.
- dict_access.
- find_element.
- set.

Uint256

See Repo
This has a struct to hold the values and 2 operations on the values

- `Uint256`
- `uint256_add()`
- `uint256_mul()`

The value is split into 2 parts high and low with
Low = least significant u251, High. =. most significant u251

We need the implicit argument `range_check_ptr` for the functions.
Functions include

- uint256_check
- uint256_add
- uint256_mul
- uint256_sqrt
- uint256_lt
- uint256_le
- uint256_unsigned_div_rem
  See the repo for others

Example of using Uint256 in Cairo

```
%builtins output range_check

from starkware.cairo.common.uint256 import (uint256_add, Uint256,
    uint256_mul)
from starkware.cairo.common.serialize import serialize_word

func main{output_ptr : felt*, range_check_ptr}():
    alloc_locals
    local num1 : Uint256 = Uint256(low=0,high=10)
    local num2 : Uint256= Uint256(low=0,high=3)
    let (local mul_low : Uint256, local mul_high : Uint256)
    = uint256_mul(num1, num2)
    serialize_word(mul_high.low)

    return ()
end
```

Boolean Expressions

From Practical Starknet

Cairo doesn't have built-in boolean expressions like `x && y` or `p || q`, but there are some tricks you can use instead. Let's say you know that `x` and `y` are each either `0` or `1`. then...

- `assert x || y` -> `assert (x - 1) * (y - 1) = 0`
- `assert !x || !y` -> `assert x * y = 0`

- `assert x && y` -> `assert x + y = 2`

Open Zeppelin cairo contracts

See Repo
In protostar you need to install the contract libraries

```
protostar install https://github.com/OpenZeppelin/cairo-contracts
```

There are some presets that can be used (Presets are pre-written contracts that extend from the library of contracts. They can be deployed as-is or used as templates for customization.)

- Account
- ERC165
- ERC20Mintable
- ERC20Pausable
- ERC20Upgradeable
- ERC20
- ERC721MintableBurnable
- ERC721MintablePausable
- ERC721EnumerableMintableBurnable

Implicit arguments recap

From cairo documentation

Builtins are predefined optimized low-level execution units which are added to the Cairo CPU board to perform predefined computations which are expensive to perform in vanilla Cairo (e.g., range-checks, Pedersen hash, ECDSA, ...).

If a function `foo()` calls `hash2()`, `foo()` must also get and return the builtin pointer (`hash_ptr`) and so does every function calling `foo()`.

Since this pattern is so common, Cairo has syntactic sugar for it, called "Implicit arguments".

```
from starkware.cairo.common.cairo_builtins import HashBuiltin

func hash2{hash_ptr : HashBuiltin*}(x, y) -> (z : felt):
    # Create a copy of the reference and advance hash_ptr.
    let hash = hash_ptr
    let hash_ptr = hash_ptr + HashBuiltin.SIZE
    # Invoke the hash function.
    hash.x = x
    hash.y = y
    # Return the result of the hash.
    # The updated pointer is returned automatically.
    return (z=hash.result)
end
```

The curly braces declare `hash_ptr` as an implicit argument. This automatically adds an argument and a return value to the function. If you're using the high-level `return` statement, you don't have to explicitly return `hash_ptr`.
The Cairo compiler just returns the current binding of the `hash_ptr` reference.

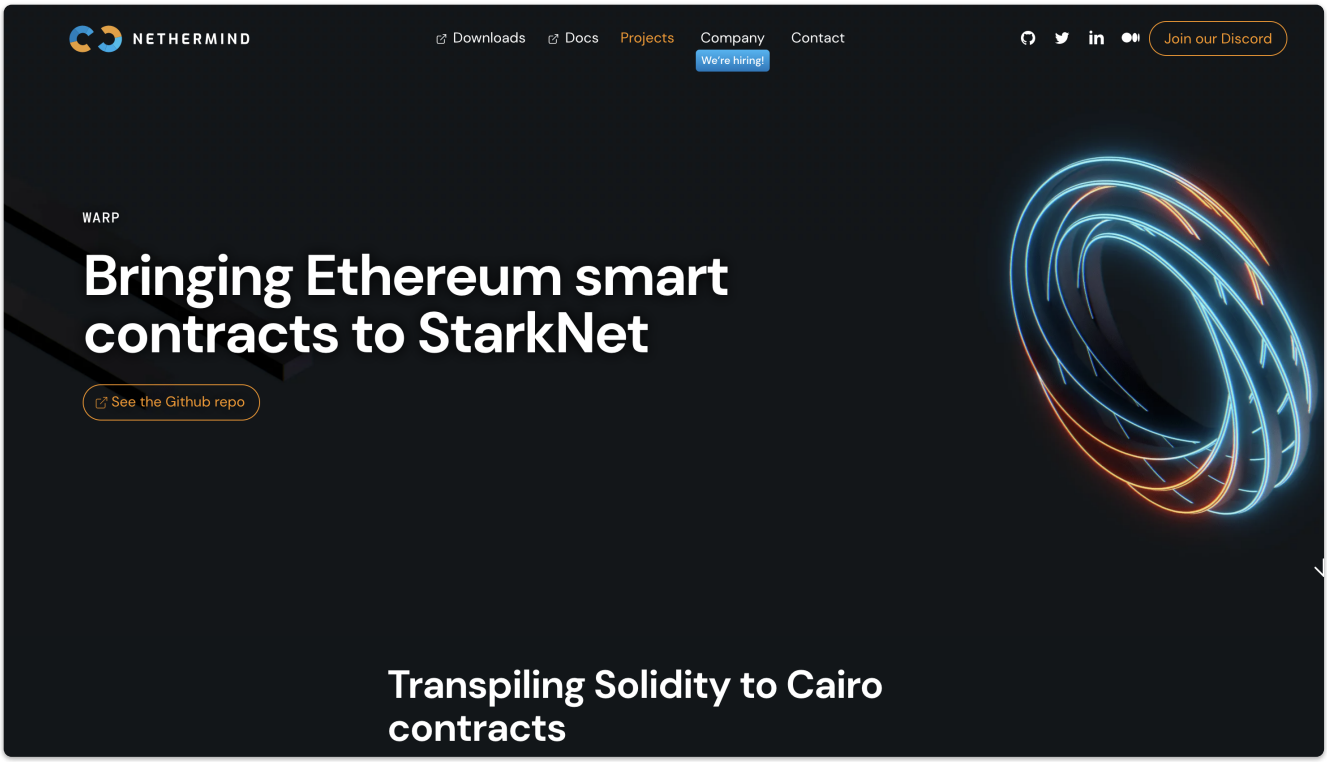
However be aware that the compiler may not be able to handle references to implicit arguments, i.e. it could be a revoked reference as we have seen elsewhere.

But according to Henri Lieutaud
"It is on our features list. Specifically:

- Removing implicit arguments from functions declarations
- Better handling revoked references with IFs etc (you still need them as soon as you have an if, for implicit arguments amongst other things)"

So implicit arguments could soon be a thing of the past.

Warp



Warp allows you transpile Solidity contracts into Cairo

Installation Instructions

See Warp installation instructions

1. On macos:

```
brew install z3
```

2. On ubuntu:

```
sudo apt install libz3-dev
```

Make sure that you have the `venv` module for your python installation.

Installation

Without any virtual environment activated run the following in order:

```
yarn global add @nethermindeth/warp
```

Run the following to see the version and that it was installed:

```
warp version
```

Finally run the following to install the dependencies:

`warp install`

Test installation works by transpiling an example ERC20 contract:

`warp transpile example_contracts/ERC20.sol`

Using Docker

`docker build -t warp .`

`docker run --rm -v $PWD:/dapp --user $(id -u):$(id -g) warp transpile example_contracts/ERC20.sol`

Using Warp

`warp transpile example_contracts/ERC20.sol`

`warp transpile example_contracts/ERC20.sol --compile-cairo`

You can then deploy your cairo code to the network, with the following commands you need to specify the network, in our case alpha-goerli

```
warp deploy test.json --network alpha-goerli
```

```
Deploy transaction was sent.
Contract address:
0x0403bd2f0abdd765398d6a50ff89cfe9ac48760f3b94ba2728bfbacdaff9f59a
Transaction hash:
0x32ca42d1341703cc957845ea53a71b3eb2e762ff148cb9dc522322eede94b65
```

You can invoke a transaction on your contract

```
warp invoke --program test.json --address
0x0403bd2f0abdd765398d6a50ff89cfe9ac48760f3b94ba2728bfbacdaff9f59a  --network
alpha-goerli --function store --inputs [13]
```

```
Invoke transaction was sent.
Contract address:
0x0403bd2f0abdd765398d6a50ff89cfe9ac48760f3b94ba2728bfbacdaff9f59a
Transaction hash:
0x1d1ec8278ccf41452737e80a54e7626299e598528363ced7a527d810f9d6881
```

And check the status

```
warp status 0x1d1ec8278ccf41452737e80a54e7626299e598528363ced7a527d810f9d6881
--network alpha-goerli
```

which will give a answer similar to

```
        {
                "block_hash":
"0x1c55254f16d087f0bf7776183c4d38549680e68600394167f304f1afe5a035e",
                "tx_status": "ACCEPTED_ON_L1"
        }
```

You should be able to see the details on the block explorer
Voyager Block Explorer