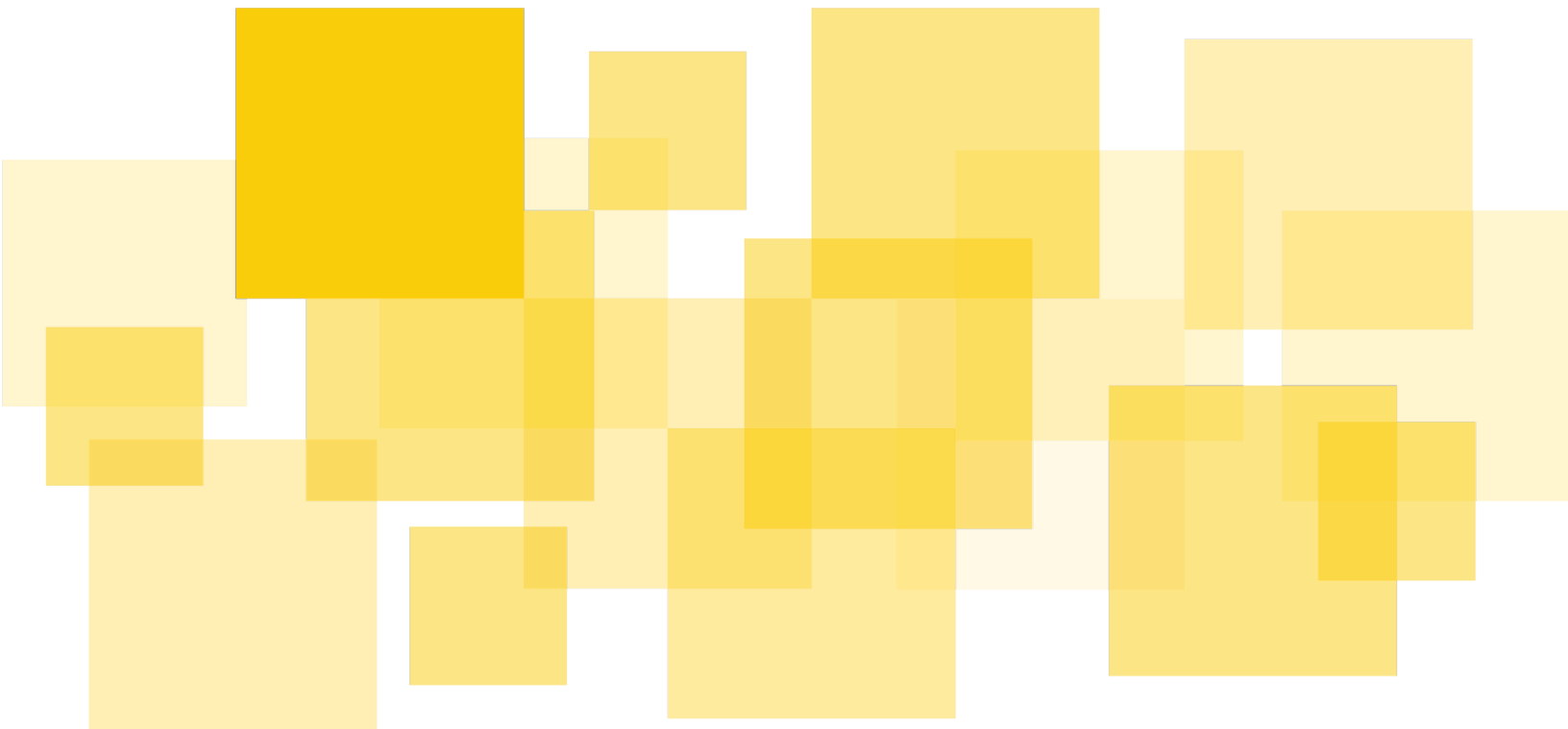


Verification Report

MultiversX Multi Asynchronous Calls

October 30th, 2023



Prepared for MultiversX by





Summary

This report presents our findings regarding the second version of the contract-to-contract asynchronous calls model within the MultiversX blockchain's protocol. Our objective was to conduct a thorough review of the protocol's design and implementation, with the aim of identifying potential issues that could lead to unexpected behavior or exploitation.

The first version of async calls, contracts were restricted to making a maximum of one async call. When a contract initiated an async call, its execution was terminated immediately. Despite this restriction, the first version allowed for multi-level async calls, permitting asynchronously called contracts to initiate new async calls.

In the second version of async calls, contracts gained the ability to make multiple async calls, each with a separate callback. Furthermore, the contract's execution can continue after initiating async calls. However, it's worth noting that the updated model no longer supports multi-level async calls.

Scope

The scope of this review is limited to the creation and execution of contract to contract asynchronous calls on the MultiversX blockchain. The review has been conducted on the following repository.

- `mx-chain-vm-go` - [69e712b5198b297dce715677bcbe27a1c7913c83](https://github.com/multiversx/mx-chain-vm-go/commit/69e712b5198b297dce715677bcbe27a1c7913c83)

The following artifacts have been reviewed.

- [asyncCall.go](#) - Defines the `AsyncCall` structure.
- [asyncCallGroup.go](#) - Defines the `AsyncCallGroup` structure, used to differentiate legacy and new async calls.
- Async context:
 - [async.go](#) - Main file defining the async context data structure.
 - [asyncComposability.go](#) - Handling of completion notifications between parent and child calls.
 - [asyncLocal.go](#) - Execution of in-shard async calls.
 - [asyncParams.go](#) - Functions for handling async call data and arguments.
 - [asyncPersistence.go](#) - Saving async contexts to contract storage.
 - [asyncRemote.go](#) - Handling of cross-shard calls.
- [hostCore](#):
 - [execution.go](#) - Entry point for synchronous and asynchronous contract calls.
- VM hooks:
 - [managedei.go](#) - Defines `ManagedCreateAsyncCall` for creating async calls, and `ManagedGetCallbackClosure` for reading callback arguments.

Besides the code review, our audit included a case analysis exploring various sync and async call scenarios involving multiple contracts. Our case analysis results and example scenarios can be accessed in our [GitHub repository](#).



Disclaimer

This report does not constitute legal or investment advice. You understand and agree that this report relates to new and emerging technologies and that there are significant risks inherent in using such technologies that cannot be completely protected against. While this report has been prepared based on data and information that has been provided or is otherwise publicly available, there are likely additional unknown risks which otherwise exist. This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system. This report is for informational purposes only and is provided on an "as-is" basis and you acknowledge and agree that you are making use of this report and the information contained herein at your own risk. The preparers of this report make no representations or warranties of any kind, either express or implied, regarding the information in or the use of this report and shall not be liable to you or any third parties for any acts or omissions undertaken by you or any third parties based on the information contained herein.

Protocol Description

Preliminaries

Call tree, root call, and leaf call

Execution of a contract call may involve making calls to other contracts, which can either be synchronous or asynchronous. These calls form a hierarchical tree structure. The root node of this tree is the initial contract call, directly initiated by a user. When this call does not spawn any additional calls, it is called a 'leaf call.'

Runtime completion vs logical completion

Runtime completion marks the conclusion of a contract's code execution, triggered by reaching the code's end, encountering an exception, or running out of gas. In cases where the contract initiates asynchronous calls during its execution, these calls are processed after runtime completion. Each call can further spawn additional calls, resulting in the generation of complex and branching call trees.

Logical completion is the point where all child calls of a contract, including any associated callbacks, are completed. A leaf call's runtime completion aligns with its logical completion since it has no children. When a contract reaches logical completion, it informs its parent call. When a parent call receives notifications from all of its child calls, it reaches logical completion.

Async context

Async context is the data structure containing information about the ongoing call. It contains the records of all the synchronous and asynchronous calls made during its execution. Regardless of the call type, every contract call has its own AsyncContext managed by the VM, serving as a node in the call tree. The AsyncContext is initialized at the start of the call's execution and populated as the execution progresses. Any sync or async calls made during this execution are recorded in the AsyncContext. After a call's runtime completion, the AsyncContext is stored in the contract's storage and remains there until all the async calls it contains are completed. It effectively keeps track of child calls during this time.

Creating an async call

Async calls are created by using the [managedCreateAsyncCall](#) VM hook. In contrast to legacy async calls, this hook only registers a new async call, allowing the caller contract to maintain its execution uninterrupted. It has the following parameters:

- `destHandle`: a managed buffer handle representing the destination contract's address.
- `value`: a big int handle for the EGLD value to be sent.
- `functionHandle`: a managed buffer handle specifying the destination endpoint name.
- `argumentsHandle`: a managed buffer vector handle for the async call arguments.
- `successOffset/successLength`: the address and length indicating the success callback function's name.
- `errorOffset/errorLength`: the address and length indicating the failure callback function's name.
- `gas`: the gas allocation for the async call's execution.
- `extraGasForCallback`: additional gas reserved for the callback execution.
- `callbackClosureHandle`: a managed buffer handle for extra arguments to be transferred from the call site to the callback function.

Conditions:

- Managed data handles must be valid.
- Offsets and lengths for callback function names must be valid.
- Callback functions must be valid, meaning they are not built-in functions, and the caller contract contains a function with the specified name.
- Multilevel async calls are not permitted; this means that neither the caller nor any of its predecessors can be async calls or callbacks. Async calls and callbacks are not allowed to make further async calls.

Any violation of the above conditions results in an error. With this information, a new async call is created and registered in the async context. After the runtime completion of the caller, async calls are processed.

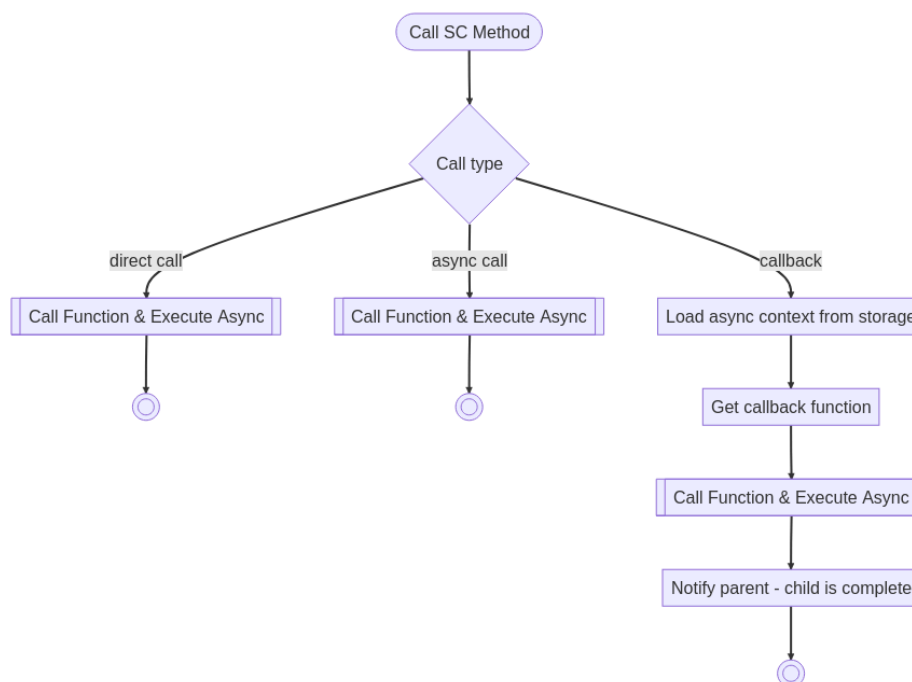
While the legacy async call creation functions, `asyncCall` and `managedAsyncCall`, remain available, it's important to note that multi-level async calls are no longer supported. This change may impact some existing contracts that rely on this feature.

Additionally, it is not allowed to combine legacy and V2 async calls within the same context.

Contract call execution

Contract calls are categorized based on their origin and creation process, with three primary types:

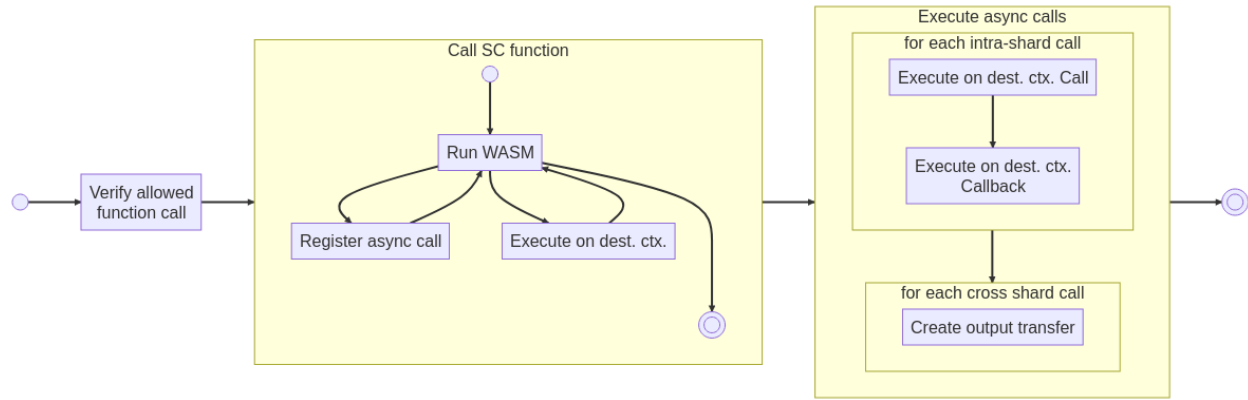
- `DirectCall`: These are explicit contract calls initiated directly by users or synchronous calls from other contracts.
- `AsynchronousCall`: These are contract-to-contract calls initiated through async call creation VM hooks, each paired with an associated callback for execution upon completion.
- `AsynchronousCallback`: These occur as a result of completing an async call, and involve executing the associated callback method with the async call's outputs.



In this section, we will delve into the execution process of contract calls, examining how different call types are processed. The core of the execution flow for all call types is encapsulated in the [callFunctionAndExecuteAsync](#) function, which we will discuss

in detail in the upcoming subsection. Subsequently, we will explore the distinct features and behaviors of direct calls, asynchronous calls, and asynchronous callbacks.

Call Function & execute async subflow



The graph above illustrates the core workflow of executing smart contract method calls. This flow is common among direct calls, async calls, and callbacks.

Initially, the [verifyAllowedFunctionCall](#) function assesses whether the requested function is not "init," "upgrade," or "callback" (unless the call type is `AsynchronousCallback`). These functions cannot be invoked directly.

Following these initial evaluations, the smart contract's WASM code begins its execution. Throughout this process, the contract may perform synchronous calls or it may register asynchronous calls. Synchronous calls can be nested arbitrarily, and each of them can, in turn, register async calls.

Upon the conclusion of the smart contract function's execution, the system proceeds to process registered asynchronous calls if there are any.

Direct calls

A direct call is a contract call originating from a user or a synchronous contract-to-contract call. It has the flexibility to initiate synchronous calls as long as it has enough gas. If a direct call, for instance, C1, initiates a synchronous call, C2, the execution of C1 is temporarily paused while C2 is processed. Once C2's execution is completed, C1's execution resumes. If C2 creates any asynchronous calls, they are processed at the end of C2 before resuming the execution of C1. Among these asynchronous calls, local calls are executed immediately, while cross-shard calls are

packaged into output transfers for delivery to their respective destination shards following the runtime completion of the root call.

A direct call can create async calls if none of its ancestors on the call tree are async calls or callbacks. When a direct call finishes executing successfully, asynchronous calls it's generated are subsequently processed. Intra-shard async calls are executed immediately, while cross-shard calls are transmitted to the shards where their callee resides. However, if the direct call fails, all the registered async calls are canceled.

Let's consider the following scenario with three contracts, where a user invokes C1's 'method1.'

```
contract C1 {  
  fn method1() {  
    async(C2, method2, cbA)  
    async(C3, method3, cbB)  
    compute()  
  }  
  
  fn cbA() {}  
  fn cbB() {}  
}  
  
contract C2 {  
  fn method2() {}  
}  
  
contract C3 {  
  fn method3() {}  
}
```

Following the creation of these async calls, C1 proceeds with various computational tasks. When C1 completes its tasks and reaches runtime completion, the asynchronous calls are processed. The specific processing of these calls depends on the shards where C2 and C3 are located.

Intra-shard async calls

Following the runtime completion of the caller call, intra-shard (local) asynchronous calls are executed using `ExecuteOnDestContext`. If there are multiple local async calls, they are executed in the order in which they were registered, and the callback associated with each async call is executed immediately after the call itself.

In the previous scenario, if all the contracts reside within the same shard, the async calls and their corresponding callbacks are executed in the following order, right after the runtime completion of C1:

1. Execute **C2.method2** (C2's logical completion)
2. Execute **C1.cbA**
3. Execute **C3.method3** (C3's logical completion)
4. Execute **C1.cbB**
5. C1's logical completion

Although async calls and callbacks are local, they can't trigger more async calls. They can only make synchronous calls.

Cross-shard async calls

After processing local async calls, the VM generates an output transfer for each cross-shard async call, containing all necessary information for its execution on the destination shard. Upon the completion of the root call's execution, all output transfers are transmitted to their respective destination shards through Metachain.

Suppose the contract C2 in the previous scenario is on a different shard. The following events occur during the handling of the async calls:

1. Execute **C3.method3** (C3's logical completion)
2. Execute **C1.cbB**
3. Create an output transfer for **C2.method2**
4. There are still unfinished child calls. Save the context in storage.
5. Send output transfers to C2's shard via Metachain

Async calls (cross-shard)

When the destination shard receives the output transfers, it sequentially processes the async calls using the `callFunctionAndExecuteAsync` function. Async calls can make synchronous calls to other contracts but are restricted from initiating additional async calls. As a result, the part of the `callFunctionAndExecuteAsync` flow related to async call processing is bypassed. Subsequently, the outcomes of these executed async calls are dispatched to the respective shards where the caller contracts reside, through Metachain.

Continuing from the previous example, the following steps take place on C2's shard:

1. Execute **C2.method2** (C2's runtime completion)

2. No async calls (C2's logical completion)
3. Send smart contract result to C1's shard via Metachain

Whether the async call is successful or results in an error, it conveys a smart contract result to the calling contract and triggers the execution of the callback.

Async Callbacks

When the caller's shard receives the smart contract result, it re-establishes the async context from the caller contract's storage. The callback function tied to the async call is then invoked with the async call output contained in the smart contract result. Similar to async calls, callbacks have the capability to make synchronous calls but are restricted from initiating further async calls. The callback function accesses the callback closure data using the [ManagedGetCallbackClosure](#) VM hook. Once the callback is complete, the caller contract is informed of the child call's completion ([NotifyChildIsComplete](#)). When there are no more child async calls to await, the parent call reaches logical completion. If the parent call is not the root call and has a parent call, it proceeds to notify its parent.

Our example scenario finally concludes with the callback execution:


1. Read the async context from storage
2. Execute `C1.cbA`
3. Notify C1
4. C1's logical completion

Properties

This section summarizes the essential characteristics of asynchronous calls. It starts with the fundamental assumptions and then presents the key properties.

Assumptions

- The inter-shard communication protocol preserves the order and integrity of messages.
- Async contexts stored in storage remain unaltered outside async call execution.
- An account can only exist in one shard.



Important properties

- Contracts can initiate async calls without restrictions based on sharding, enabling calls to contracts on the same or different shards.
- Contracts can initiate multiple async calls, each associated with different callbacks.
- Any call, regardless of its call type, can initiate synchronous calls.
- Async calls and callbacks are restricted from initiating additional async calls, preventing the occurrence of multi-level asynchronous calls.
- All calls eventually reach logical completion.