How it works > Canister smart contracts

Canisters

What are canisters

On the Internet Computer smart contracts come in the form of canisters. These are computational units which bundle together code and state. Canisters expose endpoints which can be called both by other canisters and by parties external to the IC, such as browsers or mobile apps. There are two types of endpoints. Updates are calls that can modify the state of the canisters and queries which cannot do that. A good mental model for these is that updates are used to write to the state of a canister and queries are used to read from that state. The code of a canister consists of a WebAssembly (Wasm) module. The state consists of the usual heap of the Wasm module, together with stable memory, a special type of memory which plays an important role in the life-cycle of a canister.

How do canisters work

Canisters behave much like actors from the actor-based concurrency model. Their code is single threaded and is executed in complete isolation of other canisters. Canisters communicate with one another via asynchronous messaging. When processing a message, a canister can make changes to its state, send messages to other canisters, or even create other canisters. Unlike in the traditional actor model, communication is bidirectional. Canister messages are either requests or replies. For each request sent, the IC records a callback to be invoked when the callee sends back a response. If the IC determines that there is no way for the callee to respond then the system will produce a response. Another novel aspect of the canister based model is the interplay between message processing and canister trapping. While processing a request a canister may send requests to other canisters and wait for (some of) the replies, before producing a reply to the original request. If a canister traps, its state is rolled back to the point right after it made the last outgoing call.

Resource charging

As they execute, canisters use resources in the form of memory, computation and network bandwidth. On the IC all of these are paid for using a unit called ***cycles***. To this end, each canister has a local cycles account from which the system deducts cycles as execution proceeds. Charging for memory usage is straightforward. The system keeps track of the memory used by the canister and regularly charges the canister's balance. For efficiency, this charging happens at regular intervals but not every round. In contrast, charging for computation at the time that computation is performed. To this end, the canisters are instrumented with code that allows the IC to count the number of instructions executed while processing a message. Every round, there is an upper bound on the number of instructions that can be executed during that round. If this number of instructions is exceeded, then execution is paused and continued in a subsequent round. However, cycles for the computation performed during any round are already charged at the end of that round. To prevent a buggy or malicious canister from completely taking over an execution core, the total number of rounds the execution of a canister can take is also bounded.

Charging for bandwidth is also done at the moment of use. When a canister wants to send a request to another canister, the system calculates the number of cycles that sending the message costs (the cost of sending has a fixed component and a component that depends on the size of the payload) and deducts the cost from the canister's balance. Furthermore, it also deducts the cost of sending a maximal size reply from the callee since for inter-canister messages the caller pays for reply. The cycles corresponding to the difference between the maximal size and the actual size of the reply are refunded to the canister when the reply arrives.

When canisters run out of cycles, they are uninstalled (their code and state are deleted, but the rest of the information associated with the canisters are kept). To avoid that deletion happens too suddenly, canisters have associated a so-called freezing threshold. Once the canister's balance dips below the freezing threshold (through charging by the system) then the canister stops processing any new requests; replies are still being processed. The system throws an error if at any time a canister attempts to perform an action (e.g. attaching cycles or sending a message) which would result in the cycles balance dipping below the freezing threshold.

Canister management

Canisters are managed by controllers which can be users or even other canisters. The control structure of canisters could be centralized (e.g. when the controllers include some centralized entity), decentralized (when the controller is a DAO) or even non-existent, in which case the canister is an immutable smart contract. Controllers are in charge of deploying and maintaining the canisters to the IC and they are the only entities who are allowed to perform management operations on canisters. The most common such operations are deploying a canister smart contract to the IC and starting and stopping canisters. The controller of canisters can change the canister parameters, including adding and removing controllers or changing the freezing threshold.

Controllers can update the code that runs on canisters by submitting a new Wasm module which should replace the older one. By default, updating the Wasm module of a canister wipes out the Wasm memory but the content of the stable memory remains unchanged. The IC offers an upgrade mechanism where three actions are executed atomically: serializing the Wasm memory of the canister and writing it to stable memory, installing the new Wasm code and then deserializing the content of the stable memory. Of course, a canister may ensure at all times that the data that needs to be persisted across upgrades is stored in the stable memory in which case the upgrade process is significantly simpler.

This talk covers how to create canister smart contracts on the Internet Computer, how to install and upgrade their software, and how to top up canisters with cycles.