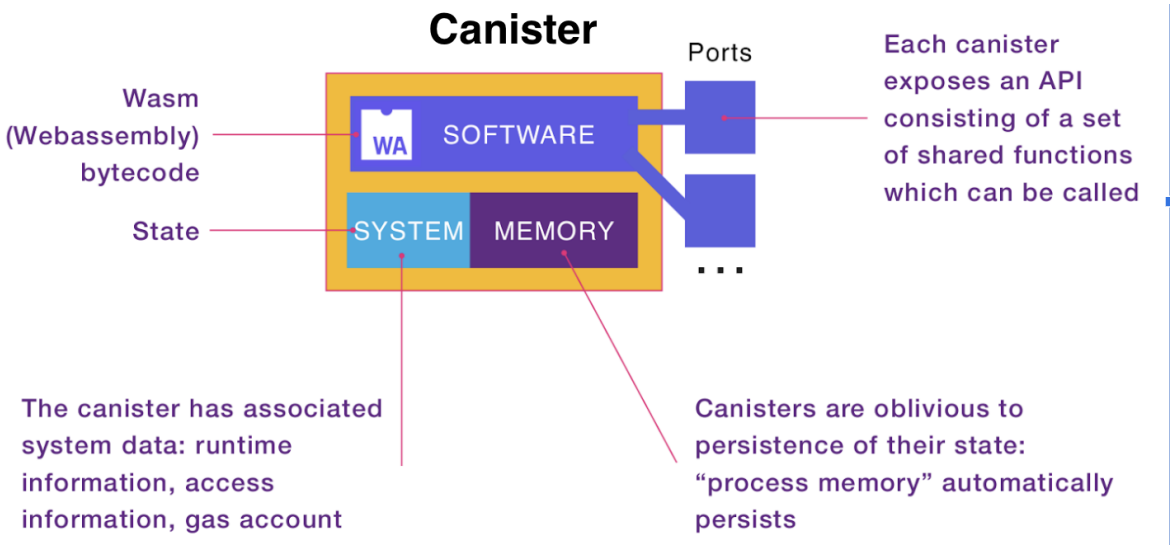
Execution

The execution layer is the topmost layer of the IC core protocol stack. It deterministically schedules and executes the messages that have been agreed on by consensus and inducted into the canister queues by message routing, thereby changing the state of the subnet in a deterministic manner on all the nodes. The execution of the same sequence of messages on every node of the subnet guarantees that the same ending state is obtained on each node of the subnet after completion of the round.

A canister smart contract on the IC consists of a Web Assembly (Wasm) bytecode representing the smart contract program and a set of memory pages representing its state. The Wasm bytecode can be modified by installing or updating the canister. The smart contract state gets modified when executing messages on the canister smart contract. Both the bytecode and the memory pages, i.e., the state, of the canister, are maintained by every node machine of the subnet the canister is installed on. Each node in the subnet holding the same canister state and ensuring that the state transitions in the same way on every node in every round is the foundation of realizing a replicated state machine and the security and resilience properties thereof that make blockchains so unique.



Replicated Message Execution

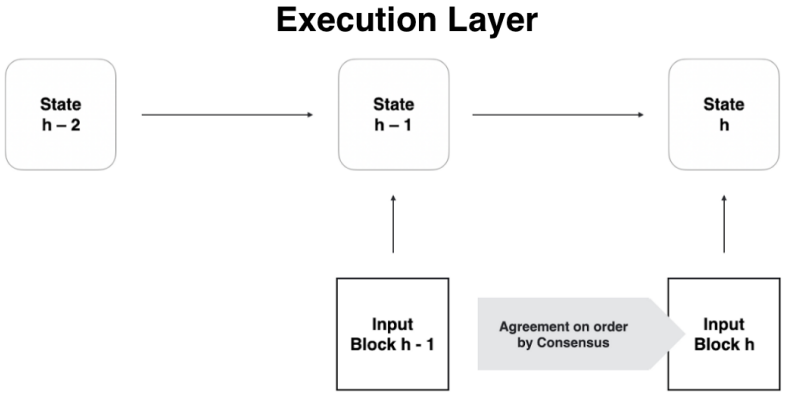
Replicated execution proceeds in rounds. In one IC round, the message routing layer invokes the execution layer once for executing (a subset of) the messages in the canister input queues. Depending on how much effort (CPU cycles) the execution of the messages of a round requires, a round ends with all messages in the queues being executed or the cycles limit of the round being reached and parts of the messages left to future rounds for execution.

Each message execution can lead to memory pages of the canister's state being modified (becoming "dirty" in operating systems terminology), new messages to other canisters on the same or different subnets being created, or a response to be generated in case of an ingress message. Changes to memory pages are tracked and corresponding pages flagged as "dirty" so that they can be processed when certifying the state.

When a message execution leads to the generation of a new canister message targeted at a canister in the local subnet, this message can be queued up directly by execution in the input queue of the target canister and scheduled in the same round or an upcoming round. This message does not need to go through consensus since the generation and enqueuing of the new message is completely deterministic and thus happens in exactly the same way on all the nodes of the subnet.

New messages targeted at other subnets are placed into the target cross-subnet queue (XNet queue) and are certified by the subnet at the end of the round as part of the per-round state certification. The receiving subnet can verify that the XNet messages are authenticated by the subnet by validating the signature with the originating subnet's public key.

The execution layer is designed at its core to execute multiple canisters concurrently on different CPU cores. This is possible because each canister has its own isolated state and canister communication is asynchronous. This form of concurrent execution within a subnet together with the capability of all of the IC's subnets executing canisters concurrently makes the IC scalable like the public cloud: The IC scales out by adding more subnets.



Non-replicated Message Execution

Non-replicated message execution, aka queries, are operations executed by a single node and return a response synchronously, much like a regular function invocation in an imperative programming language. The key difference to messages, which are also called ***update calls***, is that queries cannot change the replicated state of the subnet, while update calls can. Queries are, as the name suggests, essentially read operations performed on one replica of the subnet, with the associated trust model of a compromised replica being able to return any arbitrary result of its choice.

Analogous to update calls, queries are executed concurrently by multiple threads on a node. However, all the nodes of the subnet can concurrently execute different queries because queries are not executed in a replicated way. Query throughput of a subnet thus increases linearly with an increasing number of nodes in the subnet, while the update call performance decreases with an increasing number of nodes.

Queries are similar to read operations on a local or cloud Ethereum node on the Ethereum blockchain. A dApp should use queries for non-critical operations only. Whenever an information item to be read is critical, e.g., financial data based on which decisions are made, update calls should be used to obtain such information as the response of an update call is certified by the subnet with a BLS threshold signature and verifiable with the subnet's public key.

Deterministic Time Slicing

Each execution round progresses alongside the creation of blockchain blocks, which happens roughly once every second. This restricts how much computation can be performed in a single round, with the current limit being around 2 billion instructions given the existing node hardware.

However, the Internet Computer can handle longer tasks that need up to 20 billion instructions, and some special tasks, like code installation, can even go up to 200 billion instructions. This is acheived using a technique called "Deterministic Time Slicing" (DTS). The idea is to pause a lengthy task at the end of one round and continue it in the next. As a result, a task can span multiple rounds without slowing down the block creation rate. DTS is automatic and transparent to smart contracts, so developers don't need to write any special code to use it.

Memory Handling

Management of the canister bytecode and state (collectively memory) is one of the key responsibilities of the execution layer. The replicated state that can be held by a single subnet is not bounded by the available RAM in the node machines, but rather by the available SSD storage. Available RAM, however, impacts the performance of the subnet, particularly the access latency of memory pages. This depends a lot on the access patterns of the workload, however, – much like in traditional computer systems.

The node machines that comprise the IC are equipped with tens of terabytes of high-end SSD storage and over half a terabyte of RAM to be able to hold large amounts of replicated canister state and Wasm code and achieve good performance when accessing memory. The states obtained while executing canisters are certified (i.e. digitally signed) by the state management component of message routing. Certification of some parts of the states, including the ingress history and the messages that are sent to other subnetworks are certified every round. The entire state of a subnetwork, including the state of all canisters hosted by that subnetwork is certified once every (much longer) checkpointing interval.

Memory pages representing canister state are persisted to SSD by the execution layer, without canister programmers needing to take care of this. Having all memory pages transparently persisted enables ***orthogonal persistence*** and frees the smart contract programmers from reading from and writing to storage as on other blockchains or as in traditional IT systems. This dramatically simplifies smart contract implementation and helps reduce the TCO of a dApp and go to market faster. Programmers can always have the full canister smart contract state on the heap or in stable memory. The difference is that the heap is cleared on updates of the canister code, while stable memory remains stable throughout updates, hence its name. Any state on the heap that is to be preserved through a canister update must be transferred to stable memory by a canister programmer before an update and restored from there after the update. Best practices are that large canister state be held directly in stable memory to avoid shuffling around large amounts of storage before and after each upgrade. This also avoids the risk of exceeding the cycles limit allowed in an upgrade operation.

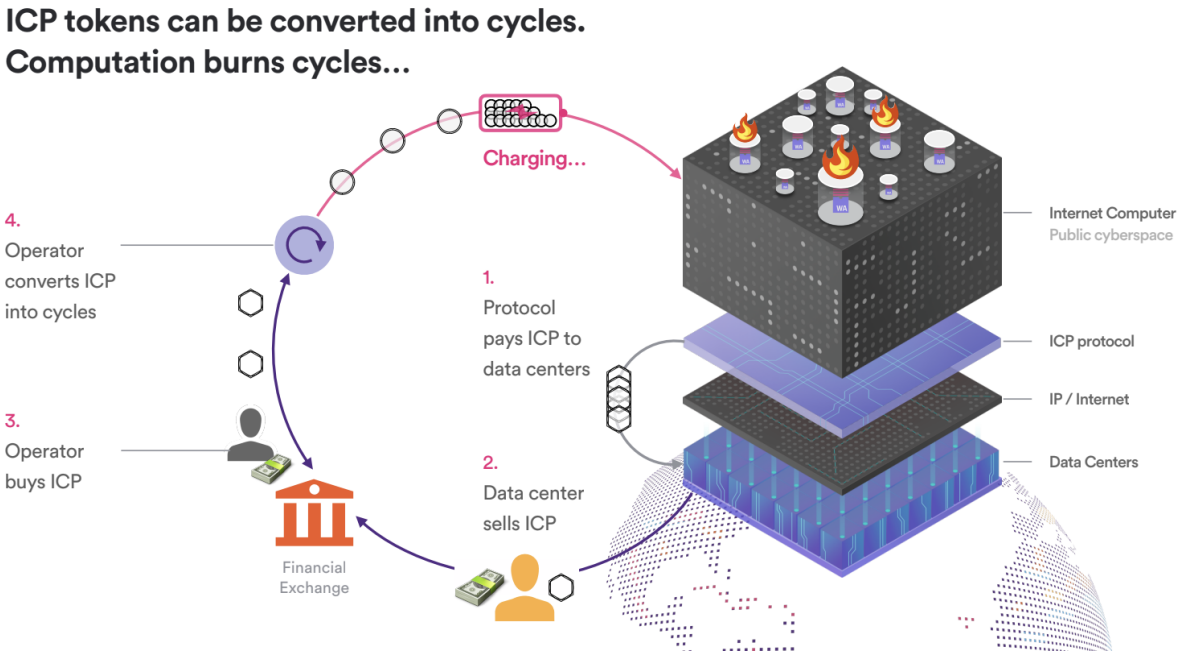
Cycles Accounting

The execution of a canister consumes resources of the Internet Computer, which are paid for with ***cycles***. Each canister holds a local cycles account and ensuring that the account holds sufficient cycles is the responsibility of its maintainer, which can be a developer, a group of developers or a decentralized autonomous organization (DAO) – users do never pay for sending messages to canisters on the IC. This resource charging model is known as ***reverse gas model*** and is a facilitator for mass adoption of the IC.

Technically, the Wasm code running in a canister gets instrumented, when the Wasm bytecode is installed or updated on the IC, with code that counts the executed instructions for smart contract messages. This allows for deterministically determining the exact amount of cycles to be charged for a given message being executed. Using Wasm as bytecode format for canister has helped greatly to reach determinism as Wasm itself is a format that is largely deterministic in its execution. It is crucial that the cycles charging be completely deterministic so that every node charges exactly the same amount of cycles for a given operation and that the replicated state machine properties of the subnet are maintained.

The memory the canister uses in terms of both its Wasm code and canister state needs to be paid for with cycles as well. Much like in the public cloud, consumed storage is charged for per time unit. Compared to other blockchains, it is very inexpensive to store data on the IC. Furthermore, networking activities such as receiving ingress messages, sending XNet messages, and making HTTPS Outcalls to Web 2.0 servers are paid for in cycles by the canister.

Pricing for a resource on the IC is extremely competitive. Prices for a given resource, e.g., executing Wasm instructions, scale with the replication factor of the subnet, i.e., the number of nodes that power the subnet.



Random Number Generation

Many applications benefit from, or require a secure random number generator. Yet, generating random numbers in the naïve way as part of execution trivially destroys determinism as every node would compute different randomness. The IC solves this problem by the execution layer

having access to a decentralised pseudorandom number generator called the ***random tape***. The random tape is built using **chain-key cryptography**. Every round, the subnetwork produces a fresh threshold BLS signature which, by its very nature, is unpredictable and uniformly distributed. This signature can then be used as seed in a cryptographic pseudorandom generator. This gives canister smart contracts access to a highly-efficient and secure random number source, which is another unique feature of the Internet Computer.