Message Routing

The Internet Computer blockchain enables users to send messages to canister smart contracts and canisters to send messages between themselves. For scalability, the Internet Computer is composed of many subnet blockchains and the Internet Computer's Network Nervous System can add new subnets as required. The message routing component routes messages to and from canisters across all of the Internet Computer's subnet blockchains and ensures that new subnets can be added seamlessly.

Message routing is the lower of the two upper layers of the protocol stack. It implements functionality that is crucial for the operation of the IC. Its responsibilities can be roughly grouped as follows:

- Induction of messages received in blocks from consensus;
- Invocation of the execution layer after successful induction;
- Routing of inter-canister messages resulting from execution within and between subnets;
- Certification of the state of the subnet;
- Synchronization of the state of the subnet to newly joining and fallen behind nodes.

Note that, although the layer derives its name from the functionality of routing messages, all the functionality listed above is equally important for the IC. Particularly, state certification and synchronization are heavily used in chain-evolution technology to enable resumption of nodes.

Message Processing

Whenever consensus produces a finalized block of messages, that is, a block that has been considered correct (notarized) and finalized by at least two thirds of the subnet's nodes, this block is handed over to message routing. This marks the transition between the lower and upper half of the protocol stack: The lower two layers are responsible for agreeing, in each round, among all nodes in the subnet on a block of messages to be executed. This block is then handed over to the upper layers for deterministic processing, which, more concretely, means passing it over to message routing which takes over the further orchestration of deterministic processing.

Once message routing receives a block of messages – recall that a block comprises both ingress messages submitted by users and XNet messages sent by canisters – the messages are extracted from the block and each message is placed into the input queue of its target canister. This process is called ***induction*** and all the queues are collectively referred to as ***induction pool***. After induction, the execution layer – the topmost layer of the core IC protocol stack – is triggered to deterministically ***schedule*** messages in the induction pool for execution and to execute them. The actual execution of messaging happens inside a sandbox, which can be thought of as a virtual machine responsible for the execution of canister messages. Message routing and execution modify the subnet state in a deterministic way, i.e., the state of the node is changed in the same way on every (honest) node of the subnet, which is crucial for achieving the replicated state machine properties of a subnet. The execution of a message can write to memory pages of the canister the message is executed on and change other metadata in the state. The execution of a message can also lead to the creation of new messages targeted at other canisters. Such a message can be either targeted at a canister on the local subnet or another subnet. In the former case, execution can directly place the new message into the input queue of the target canister. In the latter case, i.e., a new message that is targeted at another subnet, the message is placed into the so-called XNet stream for the target subnet where they can be picked up by block makers of the target subnets after the streams are certified.

Inter-Canister Messaging

As mentioned above, the execution of a canister message can lead to the creation of a new inter-canister message sent to a local or remote (on a different subnet) canister. Let us go deeper into how inter-canister messaging works.

Intra-Subnet Inter-Canister Messaging

Intra-subnet, i.e., local, inter-canister messages originating from an executing canister method do not need to go through consensus as they deterministically result from messages that have been agreed by a previous consensus round and their further execution remains completely deterministic. This holds transitively, that is, inter-canister messages can create new inter-canister messages, resulting in a tree of messages. Local message invocations can be executed as long as the cycles limit for the round has not yet been exhausted. If the cycles limit is exhausted but there are still local messages left, they will be handled in the same way as intra-subnet messages. It is important to note that this local canister-to-canister messaging is ***not***

synchronous message invocation as one might be used to from EVM-based blockchains. Rather, local messages are put into the input queue of the target canister and are scheduled for execution asynchronously. This is the standard inter-canister messaging semantics known for the Internet Computer.

Inter-Subnet Inter-Canister Messaging

Remote inter-canister messages, that is, messages sent to canisters on other subnets, are handled by routing them into the respective outgoing ***subnet stream*** for the target subnet. This routing happens at the end of the deterministic execution cycle, i.e., after execution hands back control to message routing. The XNet messages in the stream are certified (signed) using a Merkle-tree-style data representation at the end of the round by the subnet using BLS threshold cryptography as part of the per-round state certification. That is, every message in the outgoing stream is certified by the originating subnet. Replicas on the receiving subnet obtain the XNet messages during block making (part of consensus), validate the threshold signature, and include valid XNet messages in a consensus block. Thanks to using a Merkle-tree-like datastructure to encode and authenticate the XNet streams, parts of the streams can be consumed in a round by the receiving subnets and signatures can still be validated.

State Certification

The replicated state of a subnet comprises all the relevant information required for the operation of the subnet:

- Items certified per round:
  - Responses to ingress messages
  - Xnet messages to be sent to other subnets
  - Canister metadata (module hashes, *certified variables*)
- Items certified per checkpoint:
  - The entire replicated state

Certification is always done using BLS threshold signatures computed collectively by the subnet, thus certifications are computed by the subnet as a whole in a decentralized manner. The properties of the threshold signature guarantee that such a certification can only exist if the majority of the subnet agrees on the state.

State certification and secure XNet messaging enable, among others, the secure and transparent communication of canisters across subnet boundaries, a challenge that any blockchain that has multiple shards struggles with. It also provides crucial building blocks to allow users to read certified parts of the replicated state, e.g., responses to messages submitted by them. Furthermore, it allows nodes to join a subnet efficiently without replaying all blocks since genesis or fallen behind nodes to catch up to the most recent state of a subnet. All of this makes message routing an integral layer of the core IC protocol crucial for realizing some of the IC's unique and distinguishing features.

Per-Round Certification

At the end of a round, i.e., when all messages have been executed or the cycles limit for the round has been reached (to ensure rounds cannot take arbitrarily long), the message routing layer performs a certification of parts of the replicated state. A BLS threshold signature is computed to certify the part of the state tree containing

- Responses to ingress messages,
- Xnet messages to be sent to other subnets, and
- Canister metadata (module hashes, *certified variables*).

The responses to ingress messages are often referred to as ***ingress history***. The certified responses can be read and validated against the subnet's public key by users as the response to their ingress messages. Each of the public keys of the individual subnets are, in turn, certified by the NNS using the same mechanism. This means that one can verify that certified responses indeed come from the IC only using the public key of the NNS. This way of validating responses to state-changing messages to a blockchain is extremely powerful when compared to other approaches seen in the field like reading the response from a transaction log.

The per-round state certification ensures that any item of data relevant for interactions of users and subnets and between different subnets on the Internet Computer is authenticated. This particularly enables secure and verifiable inter-subnet communication, a crucial feature of the Internet Computer as well as an enabler of its scalability.

Per-Checkpoint Certification

Wasm code changed through canister updates and written-to ("dirty") memory pages of canisters and some other metadata in the replicated state do not get certified in every round. Instead they are only certified whenever a so-called ***checkpoint*** is created. A checkpoint is a copy of the replicated state that is persisted to disk. Such a checkpoint is written every multiple hundred rounds (or around 10 minutes), and for each checkpoint the subnet also computes a certification. This allows newly joining and fallen behind nodes to join in without re-executing all blocks. The state certification is done incrementally by incorporating the changes since the last checkpoint certification into the so called manifest of the previous checkpoint. The manifest can abstractly be viewed as a relatively flat Merkle tree and the incremental computation can be achieved by updating the leaves that have changed and propagating changes up the tree. Finally, the root hash of the manifest is signed with a BLS threshold signature by the subnet, thereby certifying the entire contents of the manifest. The signed result is called a ***catch-up package*** as it can be used by nodes to efficiently catch up to the point in time when the checkpoint was made. (Note that a catch-up package also contains other things required to resume, which are omitted here for the sake of simplicity). The run time of this certification operation is linear in the number of memory pages that have changed and not the overall state size on the subnet. This is crucial as a subnet can hold terabytes of state in the future and a full recertification of multiple terabytes of replicated state would not be practical at every checkpoint interval.

State Synchronization

The message routing layer implements another feature quite unique to the Internet Computer. As described above, on every checkpointing the entire subnet state is certified by the subnet through a BLS threshold signature on a Merkle-tree-like structure – the manifest – and made available as part of a catch-up package. As the name already suggests, a catch-up package allows a node to catch-up if it has fallen behind, e.g., because it was down for some time. In addition, it allows new nodes to join, e.g., if the subnet is to grow in size or a node needs to be replaced because of having been destroyed in a disaster. Such a node can download the latest catch-up package and validate its signature with the public subnet key. Once verified, the new node can download the state corresponding to the checkpoint. The downloading of the state requires the transfer of large amounts (gigabytes to terabytes) of data from the nodes's peers. This is done efficiently and in parallel from all peers, by using a protocol that chunks the state and allows for different chunks to be downloaded from different peers. Technically, the artifact transfer protocol of the P2P layer is used for transferring the state. Every chunk is authenticated through the catch-up package individually through its hash. The tree-like structure of the manifest allows to verify each of these chunks individually relative to the root hash in the catch-up package. The chunking protocol is not dissimilar to the approach that Bittorrent uses for downloading large files from many peers.

If a node is not newly added, but only had a downtime and needs to catch up, it may still have an older checkpoint. In this case, only the chunks different to the local checkpoint need to be downloaded, which can significantly reduce the to-be-transferred data volume. Efficiently computing the diff between a new state that is to be fetched and the locally available state can again make use of the Merkle-tree-like structure of the manifest. That is, one can diff the manifest to find out the chunks of the state that differ.

Once the full state corresponding to the checkpoint has been authentically downloaded, the node catches up to the current block height by processing all the blocks that have been generated in the subnet since the checkpoint and "replaying" them, i.e., executing them as it would during normal node operation, to successively make state transitions of its local state to finally reach the most recent one of the subnet.

Note that without state synchronization it might practically not be possible for nodes to (re-)join in a "busy" subnet: they would need to replay all blocks from the very first block ever created on the subnet as it is done in other blockchains. Thanks to the state sync protocol allowing to download recent checkpoints, only few blocks need to be replayed as opposed to replaying every block from the start of the blockchain. The reason why this is important is that the IC is intended to replace public cloud, i.e., to have a high throughput of operations per time unit, much like real-world cloud servers running their applications. Consider a subnet that has been running for multiple years with high CPU utilization. This would make it infeasible for a newly joining node to catch up with the subnet when trying to replay all blocks starting with the genesis block of the subnet as it would have to redo multiple CPU years worth of computation. Thus, state synchronization is a necessary feature for a blockchain that wants to operate successfully under real-world conditions where nodes do fail and need replacement.

Go even deeper

Check out the wiki page describing the message routing layer in more detail.