# Chapter 27

# Internet Computer

Many fundamental aspects of distributed systems have been introduced and discussed in previous chapters, many of which play an important role in various real-world applications.

In this final chapter of the lecture, we study a platform where many of the lecture topics come together, from broadcast and consensus all the way to state replication, which was discussed in Chapter 15.

The topic of this chapter is the *Internet Computer* (IC), which constitutes a "world computer" that extends the internet with smart contracts, providing a tamper-proof execution environment with minimal trust assumptions.[1] In this chapter, we study some of the key protocols enabling this functionality despite byzantine node behavior.

## 27.1 Canisters and Subnets

**Definition 27.1** (Canister)**.** *A* ***canister*** *is a smart contract on the IC, bundling contract logic (code) and contract state (storage). A canister exposes methods that other canisters (and users) can call by sending a message to the canister. When a canister processes a message, the canister may send messages to other canisters and message execution may change the state of the canister.*

**Remarks:**

- The IC runs state replication (Definition 15.8) for each canister.
- All canisters are hosted on dedicated individually untrusted nodes, each running the Internet Computer Protocol (ICP).

**Definition 27.2** (Subnet)**.** *A* ***subnet*** *is a set of nodes providing state replication for the canisters deployed on it.*

[1]See `https://internetcomputer.org/`.

**Remarks:**

- Each node in a subnet hosts all the canisters deployed on that subnet. There are subnets with 80,000+ canisters and subnets with just a few canisters.
- Subnets can be smaller or larger: most subnets have 13 nodes that are spread across the Americas, Europe, and Asia. For applications in need of a higher degree of decentralization, there are subnets with up to 40 nodes.
- Nodes from one subnet communicate with nodes on other subnets to deliver messages from canisters hosted on them.
- In any given subnet with $n = 3f + 1$ nodes, at most $f$ nodes may behave in a byzantine manner, cf. Theorem 17.12.
- There is one special subnet, which hosts the Network Nervous System (NNS), i.e., the governance and management canisters responsible for voting, storing node and subnet information, node provider remuneration, protocol upgrades, subnet membership changes etc.
- All nodes of the IC query the NNS canisters to learn which subnet they belong to, how to reach other nodes, and what protocol version to run.
- The IC protocol consists of four layers: Networking, Consensus, Message Routing, and Execution, described in the following sections.

## 27.2 Networking

**Definition 27.3** (IC networking layer)**.** *The **networking layer** of the IC delivers messages within a subnet **efficiently** with **bounded memory complexity**.*

**Definition 27.4** (Memory complexity)**.** *The memory complexity of an algorithm is the number of bits a correct nodes needs to store in the worst case.*

**Remarks:**

- Byzantine fault tolerant (BFT) protocol descriptions often assume networking primitives, such as best-effort and reliable broadcast.
- Practical protocols run "forever", and continuously produce messages. Guaranteed eventual delivery of all of them can require unbounded message storage for retransmission, or giving up liveness; both are undesirable.
- Many (BFT) protocols do not need such strong networking primitives.
- Protocol messages can become obsolete as protocol executions are progressing; clients only need guaranteed delivery of non-obsolete, active messages.

- Practical implementations of BFT protocols often use checkpoints, allowing them to purge obsolete messages periodically.
- The maximum number of *active* (non-aborted) artifacts is typically a function of the checkpoint interval (for example, in consensus protocols this can be measured in the number of blocks or rounds), and the number of peers. The size of each message is bounded as well.
- When protocols (i) declare obsolete messages explicitly by *aborting* their delivery, and (ii) keep the number of *active* (non-aborted) messages constant, then broadcast with bounded memory complexity can be achieved.

**Definition 27.5** (Abortable broadcast)**.** ***Abortable broadcast*** *offers two operations: broadcast and abort. For BFT protocols among $n$ nodes with a constant $C$ of* ***active*** *(non-aborted) messages at any point in time, it ensures the following properties*

- ***Abortable validity****: If a correct node broadcasts a message $m$, then every correct node eventually delivers $m$,* ***unless the sender aborts*** *$m$.*
- ***Weak Integrity****: If a node delivers a message $m$ from a correct sender $s$, then $s$ has previously broadcast $m$ (i.e., no tampering happened).*
- ***Memory boundedness****: The memory complexity of abortable broadcast is $O(n \cdot C)$.*

**Remarks:**

- In comparison with other broadcast primitives described in Chapter 18,
  - the validity property is omitted for aborted messages,
  - weak integrity is guaranteed,
  - totality and agreement are not guaranteed.
- The algorithms presented in Chapter 18 do not consider memory complexity.

**Remarks:**

- Algorithm 27.7 is based on the *slot table* data structure. A slot table is a numbered array of slots of capacity $C$, where $C$ is the bound on the number of active messages. Each slot has a version number and either is marked as free or contains a message. Each node maintains a *send side slot table $SS$* for the local active messages of the application protocol using the abortable broadcast primitive. For the reception of up to $C$ messages from peers, the algorithm manages $n-1$ *receive side slot table $RS_s$*, one for each peer $s \in [1..n]$ except itself. In summary, each node manages $n$ slot tables.

**Algorithm 27.7** Abortable Broadcast.

```
Data:
  $C$: capacity for active messages;
  $V \leftarrow 0$: send-side version number;
  $SS[1..C]$ : send-side slots with fields $version$ and $msg$
  $RS_s[1..C]$: receive-side slots with fields $version$ and $msg$ for peers $s \in [1..n]$

upon $broadcast(m)$:
  if $\nexists k \in [1..C]$ $s.t.$ $SS[k].msg = m$ then
    $i := \min\{k \in [1..C] \mid SS[k].msg = \text{none}\}$
    $V := V + 1$
    $SS[i] := \{version = V, msg = m\}$
    for each peer $s \in [1..n]$ do
      $send\_authenticated(s,\ i,\ V,\ m)$
    end for
  end if
end upon

upon $abort(m)$:
  $i := \min\{k \in [1, C] \mid SS[k].msg = m\}$
  $V := V + 1$
  $SS[i] := \{version = V, msg = \text{none}\}$
end upon

upon $receiving(s, slot, v, m)$:
  if $slot \leq C$ and $v > RS_s[slot].version$ then
    $RS_s[slot] := \{version = v, msg = m\}$
    deliver $\{s,\ m\}$
  end if
end upon

upon $periodic\ subnet\ timer\ expiry$:
  for each peer $s \in [1..n]$ do
    for $i := 1$ to $C$ do
      if $SS[i].msg \neq \text{none}$ then
        $send\_authenticated(s,\ i,\ SS[i].version,\ SS[i].msg)$
      end if
    end for
  end for
end upon
```

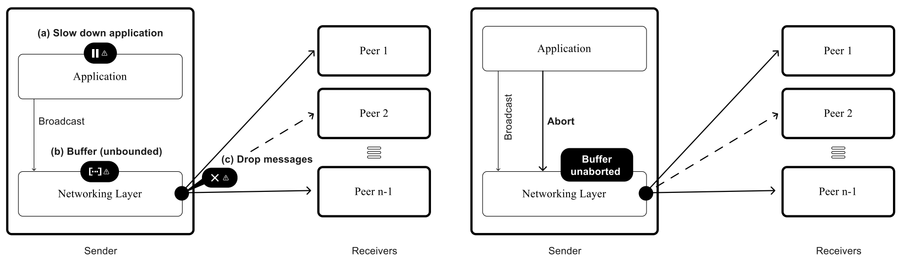

Figure 27.6: *(left)* The core challenge of traditional broadcast primitives implemented on the networking layer is reacting to backpressure, either due to slow or faulty peers or communication channels or the behavior of malicious peers. Slowing down the application (BFT protocol above), buffering indefinitely, and dropping messages are all unacceptable. *(right)* Abortable broadcast overcomes these drawbacks and can be implemented efficiently with bounds on memory and delivery guarantees.

- When the application protocol triggers a broadcast, the algorithm finds a free slot in the send side slot table (guaranteed to exist, by the condition on $C$), increments the version, and writes the message with the version in the free slot. Then it sends a single message to all peers on an authenticated unreliable channel (e.g., a signed message over UDP), announcing a new message in the slot where the version has been updated. When the client aborts a message, the corresponding slot is marked as free.

- For simplicity, we conceptually assume unbounded version numbers; in practice, using 64-bit numbers suffices to avoid rolling over.

- Abortions do not have to be explicitly announced to peers, as they will eventually be overwritten by new messages.

- When a message is received from a peer, the receiver compares the received version with the existing version in the specified slot number $\leq C$, and, if newer, delivers the message to the client.

- Figure 27.8 depicts this process with an example: Messages $A$ through $E$ were delivered from node $i$ to node $j$. Message $D$ (slot 4, version 6) as well as messages $B$ and $E$ were deleted, and message $F$ was created. Message $F$ was placed in slot number 4, and so an update message with the new message and version number 9 is sent to node $j$ (and all other peers).

- Since the messages are sent over an unreliable channel, they may get lost. The sender thus periodically retransmits all messages in its send side slot table.

**Theorem 27.9.** *Algorithm 27.7 implements abortable broadcast.*

*Proof.* The crucial observation is that the size of each slot table is bounded to at most $C$ slots. Line 2 and 18 maintain this invariant for the send and

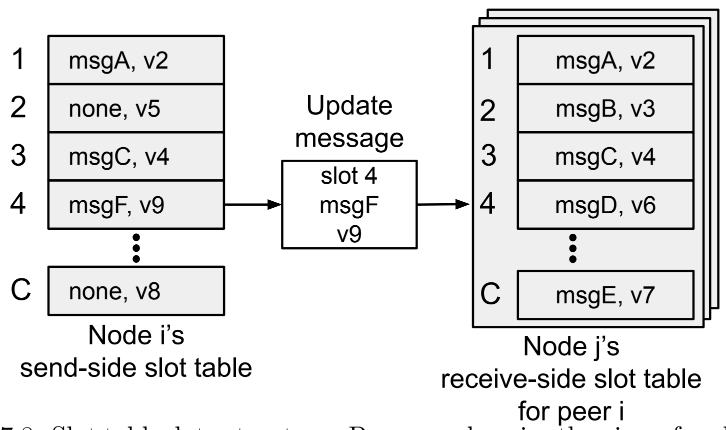

Figure 27.8: Slot table data structure. Peers synchronize the view of each other's message set in an eventually consistent protocol, with strict bounds.

receive side, respectively, and thus each of the $n$ slot tables (one for the send side and $n-1$ for the receive side) contain at most $C$ messages and version numbers. As a consequence, only the latest version of each slot is retransmitted to all peers periodically. This suffices to achieve the abortable guaranteed delivery property 27.5, while keeping the memory usage of the algorithm bounded (achieving 27.5). The integrity property 27.5 is achieved thanks to the authenticated channel. □

**Remarks:**

- Totality (see Definition 18.3) for $m$, can be achieved by adding $m$ to the send slot table upon delivery.
- Without additional mechanisms, this will make the communication complexity measuring the total number of bits sent around quadratic in the number of nodes, i.e., $\Theta(n^2 \cdot |m|)$, where $|m|$ denotes the size of message $m$, while $\Omega(n \cdot |m|)$ suffices.

## 27.3 Consensus

**Definition 27.10** (IC consensus layer)**.** *The* ***consensus layer*** *on the IC validates messages and determines an order for processing at every node in the subnet.*

**Remarks:**

- For all nodes to transition to the same state, they must process exactly the same set of messages in the same order.
- The IC consensus algorithm guarantees the standard agreement and termination properties of Definition 16.1 under the assumption that at most $f < n/3$ nodes are byzantine.

- The agreement property (if two correct nodes decide, they decide on the same set of messages) holds in the asynchronous communication model, whereas the termination property (every correct node eventually decides) holds if there are periods where all messages arrive within a bounded time, i.e., termination requires a partially synchronous communication model.
- The algorithm does not need to know this upper bound on message delays as it can increase its estimate if the current estimate turns out to be too small (because it takes longer for messages to arrive). For the sake of simplicity, we assume that the upper bound on the message delay is known and is 1.
- The algorithm makes use of sophisticated cryptographic tooling.

**Definition 27.11** (BLS signature scheme)**.** *The **BLS signature scheme** is a signature scheme, which consists of a key generation, signature generation, and signature verification algorithm with the following properties:*

- *For a given key and message, there is only one valid signature.*
- *BLS signatures can be aggregated, which means that multiple signatures on the same message can be combined into a **compact** multi-signature of the same size as a single signature.*

**Remarks:**

- Each node of the IC has a private key for the BLS signature scheme.
- Additionally, the algorithm uses a *BLS threshold signature scheme* (see Definition 19.21) with the property that *any* $f + 1$ out of $n$ *signature shares* can be combined deterministically into the *same* signature.
- The algorithm needs a source of unpredictable randomness.

**Definition 27.12** (Random beacon)**.** *The **random beacon** is a sequence of random (256-bit) numbers with the property that every consensus round yields the next number, which is unpredictable given the previous numbers.*

**Remarks:**

- The numbers are constructed using the BLS threshold signature scheme: Given a number $b$ known to all nodes, the next number $b'$ is obtained by having each node broadcast its signature share of $b$ and then constructing the unique and deterministic signature for $b$ by combining $f + 1$ received signature shares.
- Note that the $f$ malicious nodes cannot precompute this sequence because $f + 1$ signature shares are required to determine the next number (and $f$ signature shares do not provide any information about the signature).

**Definition 27.13** (Block)**.** *A **block** on the IC is a batch of canister messages, each message targeting a specific canister on the same subnet.*

**Remarks:**

- There is an empty, hard-coded *genesis block*. Each other block has a unique predecessor block.
- A non-genesis block is valid if its messages are valid (in particular, they must target canisters on this subnet and be signed correctly), and if their predecessor is valid.
- Additional block metadata and more extended validity conditions are defined later in this chapter when required for higher layer functionality.

**Definition 27.14** (Block height)**.** *Each block is associated with a non-negative number, called its* ***block height****. The genesis block has block height* 0 *and the block height of every other block is the block height of its predecessor block* $+1$.

**Remarks:**

- Every node maintains a directed tree of blocks rooted at the genesis block.

**Remarks:**

- It is implicitly assumed that all messages are signed and that any message that does not bear a valid signature is silently dropped. This rule also applies to signature shares, which are verified in the same manner as regular signatures and discarded if they are invalid.
- The algorithm pseudocode and proof use broadcast and reliable broadcast for simplicity. To use abortable broadcast instead of broadcast, a checkpointing procedure must be introduced. Checkpoints are produced at certain heights and consist of the state at this point as well as all the necessary key material. Messages for lower heights than the current checkpoint are aborted. This alone is not sufficient to bound the number of active (non-aborted) messages. To ensure the totality and agreement properties of reliable broadcast, all valid received blocks must be forwarded to peers and a mechanism to prevent equivocation (a block maker proposing two or more distinct blocks for the same height) is required.

**Definition 27.16** (Epoch)**.** *An* ***epoch*** *on the IC is a time period during which the nodes of a subnet execute the consensus algorithm to agree on at least one block for a specific block height.*

**Remarks:**

- In other words, the consensus algorithm is executed exactly once per epoch. Each epoch lasts multiple communication rounds. As we will see, it is possible that there are multiple blocks in the same epoch $h$.
- For each epoch, a *block maker* is chosen using the random beacon: The random beacon $b_h$ of epoch $h$ is used as the seed of a pseudo-random function to determine a random permutation of the $n$ nodes for this epoch. The permutation defines a unique rank $r \in \{0, \ldots, n-1\}$ for every node.

**Algorithm 27.15** IC Consensus: Actions at node $v_i$ for epoch $h$

```
1: $r_i$ := rank($i$, beacon($h$))
2: if $\geq 2r_i + \varepsilon$ time passed and tree_height() $< h$ then
3:   $B$ := build_block($h$)
4:   Reliable-broadcast $block(B, i, h)$
5: end if
6:
7: upon receiving $block(B, j, h)$ for the first time:
8:   $r_j$ := rank($j$,beacon($h$))
9:   if $\geq 2r_j + \varepsilon$ time passed and tree_height() $< h$ and is_valid($B$)
     then
10:    $n_i^B$ := notarization_share($B$)
11:    $nc_i^h := nc_i^h + 1$    // Count the number of different notarization shares
12:    $id^B := \mathcal{H}(B)$    // A block hash is used as the identifier
13:    Broadcast $notarization\_share(id^B, n_i^B, h)$
14:  end if
15: end upon
16:
17: upon receiving $notarization\_share(id^B, n_j^B, h)$ for the first time:
18:   $N_i^B$ := notarization_shares($id^B$)
19:   $N_i^B := N_i^B \cup \{n_j^B\}$
20:   if $|N_i| \geq n - f$ and $B$ received but not in tree then
21:     add_to_tree($B$)    // The height is now at least $h$
22:     if $nc_i^h = 1$ then
23:       $F_i^B$ := finalization_share($B$)
24:       Broadcast $finalization\_share(id^B, F_i^B, h)$
25:     end if
26:     $b_i^{h+1}$ := beacon_share(beacon($h$))
27:     Broadcast $random\_beacon\_share(b_i^{h+1}, h + 1)$
28:   end if
29: end upon
30:
31: upon receiving $finalization\_share(id^B, f_j^B, h)$ for the first time:
32:   $F_i^B$ := finalization_shares($id^B$)
33:   $F_i^B := F_i^B \cup \{f_j^B\}$
34:   if $|F_i^B| \geq n - f$ and $B$ in tree then
35:     finalize($h, B$)    // End all epochs $\leq h$
36:   end if
37: end upon
38:
39: upon receiving $random\_beacon\_share(b_j^{h+1}, h + 1)$ for the first time:
40:   $RB^{h+1} := RB^h \cup \{b_j^{h+1}\}$
41:   if $|RB^{h+1}| = f + 1$ then
42:     build_beacon($RB^{h+1}, h + 1$)    // Start epoch $h + 1$
43:   end if
44: end upon
```

- A node with rank $r \in \{0, \dots, n-1\}$ is allowed to make a proposal after $2r + \varepsilon$ time from the start of the epoch, where $\varepsilon > 0$ is an arbitrarily small constant.

- When node $v_i$ receives a block $B$ for epoch $h$ from some node $v_j$ and at least the required amount of time has passed based on $v_j$'s rank and there is no block at height $h$ already, then $v_i$ checks the validity of the block and generates a so-called *notarization share* $n_i^B$ for block $B$ using its BLS signing key and broadcasts it. A notarization share by node $v_i$ means that $v_i$ validated the block, i.e., the block is a valid continuation of a notarized predecessor block from epoch $h - 1$, and is an endorsement of this block for epoch $h$.

- Once the set $N_i^B$ contains $n - f$ notarization shares, the shares can be combined into a (compact) multi-signature, proving the notarization of the block, which is to be understood as "the whole subnet considers block $B$ valid for epoch $h$". Subsequently the block is added to the tree. In other words, the tree contains all notarized (and thus valid) blocks a node is aware of.

- If this is the only block for which $v_i$ has ever broadcast a notarization share, $v_i$ broadcasts a so-called *finalization share*, which is simply another BLS signature on block $B$. A finalization share from a correct node $v_i$ says that $v_i$ guarantees that it never endorsed any block for epoch $h$ other than $B$.

- If at least $n - f$ finalization shares are received, the epoch $h$ is marked as *finalized*. The function `finalize` not only finalizes epoch $h$ with $B$ being the unique block of this epoch but it also recursively finalizes all epochs $h' < h$, defining the unique predecessor of $B'$ of $B$ as the finalized block of epoch $h - 1$ and so on. At this stage, the node no longer responds to messages of any epoch $h' \leq h$.

- The next epoch is started when $f + 1$ signature shares for the next random beacon are locally available.[2]

**Remarks:**

- Figure 27.17 shows an example block tree. There may be multiple (notarized) blocks generated in the same epoch; however, there can only be one finalized block for a certain epoch $h$. Since there is only one block for this epoch and each block has only one predecessor, the blocks for all epochs lower than $h$ can be finalized as well. Any forks at block heights lower than $h$ can be discarded.

**Lemma 27.18.** *For every epoch $h$, if correct nodes $v$ and $v'$ finalize blocks $B$ and $B'$, then $B = B'$.*

*Proof.* Assume for the sake of contradiction that $B \neq B'$ and that $f^* \leq f$ nodes did not behave according to the protocol. Since $v$ finalized $B$, it must

[2]Note that the random beacon is actually constructed at the beginning of the epoch on the IC. The random beacon is constructed at the end of the epoch here for ease of exposition.

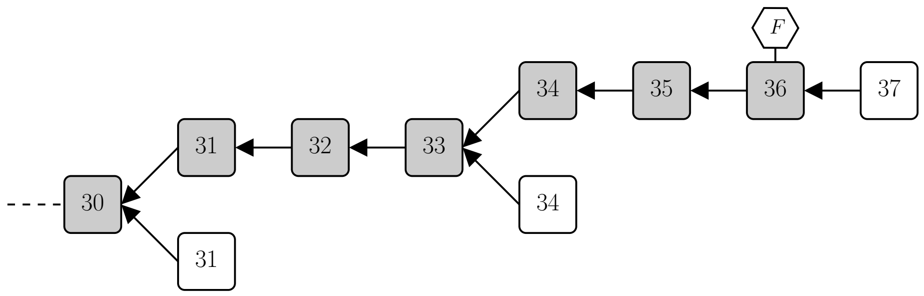

Figure 27.17: A possible block tree with forks in epochs 31 and 34. Since a block was finalized in epoch 36, all its predecessors are implicitly finalized as well.

have received $n-f$ finalization shares and therefore a set $S$ of at least $n-f-f^*$ finalization shares from correct nodes. The same argument applies for block $B'$, i.e., $v'$ must have received a set $S'$ of at least $n-f-f^*$ finalization shares from correct nodes.

The union of $S$ and $S'$ can be at most the set of all nodes that behaved correctly, i.e., $|S \cup S'| \leq n - f^*$. Moreover, a correct node only sends the finalization share for at most one block, which implies that $S$ and $S'$ must be disjoint. We get that

$$n - f^* \geq |S \cup S'| = |S| + |S'| \geq 2(n - f - f^*),$$

which implies that $3f \geq 2f + f^* \geq n$, a contradiction. □

**Lemma 27.19.** *For every epoch $h$, at least one block is notarized.*

*Proof.* Let $w$ be the lowest ranked correct node of epoch $h$. Let $r_w$ be its rank. After $2r_w + \varepsilon$ time, $w$ will broadcast its constructed block unless it has already notarized a block before. In either case, reliable broadcast ensures that all correct nodes will eventually receive some block that is endorsed by all correct nodes, which implies that $v$ must eventually receive at least $n-f$ notarization shares for this block, which it can then notarize. □

**Lemma 27.20.** *For every epoch $h$, every correct node will eventually transition to epoch $h+1$.*

*Proof.* When node $v_i$ notarizes a block, it broadcasts its random beacon share $(b_i^{h+1}, h+1)$. According to Lemma 27.19, every correct node eventually notarizes a block for epoch $h$, which implies that every correct node eventually receives at least $f+1$ random beacon shares, at which point it builds the random beacon for epoch $h+1$, triggering the start of epoch $h+1$. □

**Lemma 27.21.** *If there is a period of synchronicity of duration at least 3 from the start of an epoch and the rank-0 block maker in this epoch is correct, the epoch will be finalized.*

*Proof.* Consider epoch $h$ starting at some time $t$. We assume that all messages arrive within 1 time unit in the time interval $[t, t+3]$. The correct rank-0 block maker broadcasts a block $B$ at time $t$, which every correct node receives by time $t+1$. At this point in time, each correct node broadcasts its corresponding notarization share and these shares arrive at all correct nodes by time $t+2$. Since each correct node receives at least $n-f$ notarization shares and no correct may have notarized any other block at time $t+2$ (no other block can be notarized before time $t+2+\varepsilon$), every correct node broadcasts a finalization share. Thus, each correct node receives at least $n-f$ finalization shares by time $t+3$ and finalizes the epoch. $\square$

## 27.4 Message Routing

**Definition 27.22** (IC message routing layer)**.** *The **message routing layer** of the IC ensures that messages from consensus reach their destination canister and provides authenticated information to the users.*

**Remarks:**

- A canister is hosted and executed on exactly one subnet, i.e., the set of all canisters is partitioned across all subnets.
- Each canister can be viewed as a state machine where the inputs are request messages from users or other canisters and the outputs are the response messages. Both input and output messages are stored in queues. The canister code describes the state transition when processing messages.
- Each subnet contains a set of canisters and sends/receives messages from users and other subnets, i.e., each subnet can be viewed as a state machine.
- All subnets together form the IC state machine.
- In this section we will first look at the state machine formed by a single subnet, with canisters processing user messages only, and then consider the interaction across subnets as well.

**Definition 27.23** (Ingress message status)**.** *An ingress message from a user to a canister can be in status `UNKNOWN` (starting state), `RECEIVED` (nodes agree to have received it), `PROCESSING` (message execution has started), `REPLIED` (response has been computed successfully) and `REJECTED` (system or canister decided not to continue working on this message). Messages from users, as well as their status and response are stored in the **ingress history**.*

**Remarks:**

- A message transitions from one status to the next until it is `REPLIED` or transitions to `REJECTED` from any earlier status directly.
- Users can query the ingress history of the subnet to learn about the status of their message.
- In a crash-recovery model without Byzantine behavior, users trust any node to report on the status and ingress history correctly.

**Definition 27.24** (Replicated system state, certificate)**.** *The **replicated system state** of each subnet at a given height can be represented as a Merkle tree with the state of the canisters, the input and output queues, the ingress history and the current subnet time as leaves. A **state certificate** for $h$ is a $(n-f)$-out-of-$n$ threshold-signature of the Merkle tree root hash. The highest height for which a node has a valid state certificate is called **certified height**.*

**Remarks:**

- It is essential that all of this state be updated in a completely deterministic fashion so that all nodes maintain exactly the same state.
- After the message execution phase for a given height $h$, the message routing layer will initiate the process to certify the state for $h$ by sending out threshold signature shares.
- Hashing the whole replicated system state is an expensive operation, it could take longer than the time allocated for the execution per height, therefore a subset of the state is hashed in practice.
- The node does not wait for all certification shares to arrive but continues with the next round after triggering the certification process. Therefore, the certified height can be below the height for which a node currently executes messages. This also implies that the certified height can be different for every node in a subnet and does not change deterministically like the replicated state.
- The certified state is used in several ways in the IC:
  - *Output authentication.* Users rely on the certified state to verify responses whose hashes are stored as Merkle tree leaves in the certified state. The Merkle proof consists of the hashes to verify the path to the root and the root signature.
  - *Preventing and detecting non-determinism.* Consensus guarantees that each replica processes inputs in the same order. Since each replica processes these inputs deterministically, each replica should obtain the same state. However, the IC is designed with an extra layer of robustness to prevent and detect any (accidental) non-deterministic computation, should it arise. To this end, the certified height is added to consensus blocks and a node considers a block valid only if the node's certified height exceeds the certified height in the block.
  - *Execution and consensus speed.* The certified state is also used to coordinate the execution and consensus layers: If consensus is running ahead of execution (whose progress is determined by the last height with certified state), consensus will be "throttled". That is, if the difference between the finalized height and the certified height exceeds a threshold, then the time until which consensus waits before creating and notarizing blocks is increased.

To achieve the functionalities described above, the block payload and the validity conditions used in consensus are extended.

**Definition 27.25** (Block payload)**.** *A block payload comprises*

- *messages: set of messages to canisters on this subnet*
- *$certified\_height$: state certificate for this height exists*

**Definition 27.26** (Extended Block Validity)**.** *A node considers a block valid if*

- *none of the payload messages occur in predecessor blocks,*
- *a block's certified height is at least the previous block's certified height and at most the node's certified height,*
- *all messages are signed correctly.*

**Remarks:**

- To create a valid block $B$ in `build_block()`, a node's consensus layer
  - Selects a set of ingress messages that have not occurred in any of the predecessor blocks
  - Calls message routing's `create_payload` function (see Lines 16–18 of Algorithm 27.27) to obtain the ingress history and the certified height.
- To decide if a block from another node is valid, the consensus layer can check first the locally available predecessor blocks. To verify the message routing parts of the block, `is_payload_valid` (see Lines 20–22 of Algorithm 27.27) is called, which verifies that the certified height is not above the height for which the node has a valid full certified state signature and that the ingress messages do not show up in the ingress history already.
- Once block $B$ has been finalized, blocks below $B.certified\_height$ and expired ingress messages are no longer needed to determine block validity and can be discarded.

**Remarks:**

- The message routing layer inserts messages from blocks in one of multiple input queues, one per canister.
- For each block height, the execution layer will consume some of the messages in the input queues and update the ingress history and the replicated state of the relevant canisters.
- All message routing steps in `process_payload()` and `is_payload_valid()` must be fully deterministic. Therefore, execution time is not measured in seconds, but in low-level machine language instructions.
- The execution of a message may fail. For ingress messages, the ingress history is updated to status `REJECTED` in this case. If a response to the message has been generated successfully, the ingress history status is set to `REPLIED`, allowing the user to collect the response.

**Algorithm 27.27** Message Routing: Processing ingress messages

```
upon process_payload($payload$) called:
    $state.height := state.height + 1$
    insert_into_input_queues($payload, state$)
    for $m \in payload$ do
        $state.ingress\_history[m]$ := RECEIVED
    end for
    while execution time left do
        $m$ := pop_from_input_queues($state$)
        $state.ingress\_history[m]$ := PROCESSING
        execute($m, state$)
    end while
    prune_ingress_history($state$)
    trigger_certify_state($state$)    //sets $certified\_height$ eventually
end upon

upon create_payload() called:
    return ($state.ingress\_history, certified\_height$)
end upon

upon is_payload_valid($B$) called:
    return $B.certified\_height \leq certified\_height$ and
      $B.ingress\_payload \cap ingress\_history = \emptyset$
end upon
```

- `prune_ingress_history` removes every message $m$ from the ingress history if the message status is a terminal state and the message expiry is at least $GRACE\_PERIOD$ in the past, i.e., $cur\_time - m.expiry > GRACE\_PERIOD$. The grace period gives users enough time to fetch the status and response of their messages and enables bounding the period a message occupies memory in the system.

- The creation and collection of threshold signature shares on the per-height certified state is started by `trigger_certify_state`. When the full signature of a state height greater than $certified\_height$ has been collected, $certified\_height$ is updated. Note that the next finalized block may be submitted to message routing before a full signature on the current state has been collected.

- The consensus layer is decoupled from the message routing and execution layers in the sense that only messages from finalized blocks of the chain reach message routing and execution. Temporary block tree branches are pruned before their payloads are passed to message routing and execution. This is in contrast to other blockchains that execute blocks speculatively, before ordering and validating them.

To let the canister state machines interact with each other, regardless of whether they reside on the same or different subnets, queues and streams are introduced and the Consensus and Message routing layer are extended to process messages from other subnets.

**Definition 27.28** (Canister queues and streams)**.** *For each canister $C$ there is a separate **input queue** for each other canister $C'$ from which $C$ receives messages and there is one queue for user-generated messages to $C$. For each canister $C''$ for which $C$ creates messages there is an **output queue**. Messages for other subnets are ordered into **streams**, one for each subnet that canisters communicate with.*

**Remarks:**

- $C'$ and $C''$ may reside on the same or on different subnets.
- `process_payload` in Algorithm 27.27 is adapted as follow:
    - The ingress history is only modified for messages from users.
    - In addition to responses to user messages, message execution may create new messages to the same or other canisters, which are put in the corresponding output queues. If a canister needs to wait for their responses, the status remains `PROCESSING`.
    - After the execution loop the message routing layer takes the messages in the output queues, organizes them into subnet-to-subnet streams and exchanges them with nodes in other subnets.
    - Communication across subnets is referred to as *xnet (cross-net) communication*, taking place at the end of processing the payload.
    - Creating and validating payload of Consensus and Message routing can be extended to include messages from canisters on other subnets (xnet messages).
    - When exchanging messages with other subnets, individual message signatures are constructed and verified with the Merkle tree paths and the certification signature of the Merkle tree root hash. For xnet communication a node sends and receives a selection of messages to and from nodes on other subnets, respectively. Since all the signatures are based on a threshold of $n - f$, a malicious node cannot convince a correct node to accept an invalid message. This fact is also used when checking subnet signatures for messages from other subnet.
    - `is_payload_valid(`$B$`,` *previous_xnet*`)` requires the xnet messages since $B.certified\_height$ to be available to check if no xnet messages have been skipped in the stream.
- `create_payload` and `is_payload_valid` are modified to also return and check xnet messages, respectively. The function `is_payload_valid(`$B$`,` *previous_xnet*`)` requires the xnet messages since $B.certified\_height$ to be available to check if no xnet messages have been skipped in the stream. This input needs to be provided by Consensus.
- The whole message routing process taking place for each finalized block is visualized in Figure 27.29.

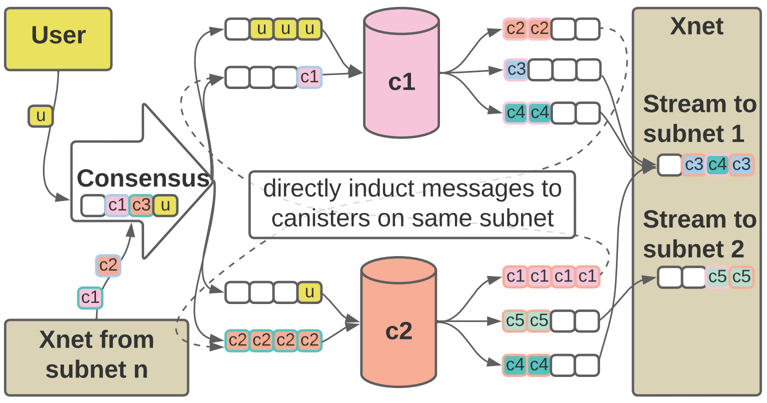

Figure 27.29: Routing messages through the IC protocol stack. Messages for canisters, issued by users or canisters on other subnets, are validated and ordered by consensus. Subsequently, messages are put into the input queues of their destination canisters. Messages created by canisters are put into output queues from where they are either transferred to their respective input queues on the same subnet (bypassing consensus) or sent as part of streams to their target subnet.

## 27.5 Execution Environment

**Definition 27.30** (IC execution environment layer)**.** *The **execution environment layer** of the IC schedules and processes canister messages.*

**Definition 27.31** (Request, response)**.** *Methods exposed by canisters can be called by other canisters (and users) by sending a **request** message. After executing the method with the request message, the method then provides a **response** message to the caller, which the caller can process. When processing a message (either a request or a response), a canister can change its state and issue further calls to itself or other canisters.*

**Remarks:**

- Only a single message is processed at a time per canister. Thus message execution is sequential and never parallel per canister.
- Different canisters can process messages in parallel.
- Whenever a canister issues a downstream request, the execution of the upstream call is effectively suspended until the response arrives, but the canister is allowed to process other messages (both other requests and responses).
- Multiple messages from different calls can be interleaved and have no reliable execution ordering.
- In case of traps or panics, the state changes are reverted.

- Message delivery between canisters is asynchronous. Successfully delivered requests are received in the order in which they were sent.

- It is important for many applications to have a message expiry mechanism. This facilitates user-side decisions on whether another message may be submitted, without risking to have both of them executed. E.g., if a message for a transfer of 100$ to another user has expired, the user can resubmit a transfer and will not be debited twice.

To achieve unique execution before expiry, the block payload and validity as well as message routing and execution must be modified.

**Definition 27.32** (Block time and validity, subnet time)**.** *A block payload contains time, representing the timestamp at block creation. Block validity is extended to also comprise the following conditions.*

- *The block's time is higher than the previous block's time and at most the node's current local system time,*

- *the expiry time of all ingress messages exceeds the time in the block, and*

- *the expiry time of all ingress messages is at most max_expiry_interval greater than the time in the block.*

*A subnet's time at height $h$ is defined as the time in block from round $h$.*

**Remarks:**

- To create a valid block $B$ in `build_block()`, the consensus layer of a node sets $B.time$ to its current local system time.

- As long as all nodes are synchronized well, this guarantees correct nodes can create and validate blocks from other correct nodes.

- In practice it happens that node providers do not set up their nodes correctly which prevents them from synchronizing properly and thus from participating in consensus (less than 1% of the nodes were ever affected by this).

- Time is added to the replicated system state at every height and certified. Thus each subnet has its own time.

- Execution of an ingress message is only started if the current subnet time is below the ingress message's expiry.

**Theorem 27.33** (Unique execution before expiry)**.** *Every ingress message with an expiry field will either enter the state `PROCESSING` in Algorithm 27.27 exactly once before its expiry time with respect to the subnet's time or it will never be processed. Subnet-signed responses guarantee that subnet time is strictly monotonic and that the reported ingress history status transitions have occurred at $f + 1$ or more correct nodes.*

*Proof.* Correct nodes adhere to the block validity conditions and only create and notarize blocks with a higher timestamp than previous blocks. Since $2f + 1$ notary shares are necessary for a full notarization, at least $f + 1$ honest nodes must be involved in a successful notarization and thus only blocks with monotonically increasing time will be notarized. Any finalized block has been notarized by $n - f$ nodes, hence it holds for any subnet with at most $f$ byzantine nodes that the subnet time is strictly monotonic. By the same argument, only ingress messages with expiry time in the future are in finalized blocks and an ingress message can appear at most once in blocks. To this end, the consensus layer checks the *ingress_payload* in all the notarized predecessor blocks with time above the newest block time minus *max_expiry_interval*. Thus, a correct node will not create or notarize a block that contains duplicate ingress messages.

After its expiry, the message cannot make it into the processing state because time is checked again before execution is started on correct nodes.

$2f + 1$ signatures are necessary to certify the replicated system state, so up to $f$ byzantine nodes cannot make users or other nodes turn back the subnet time or believe an ingress message is in a different state than $f + 1$ correct nodes considered valid at some point. □

**Remarks:**

- This theorem is important for applications with asset transfers. For example, a user wants to be sure that if they issue a message for a transfer of some tokens to another user, it will not be executed twice (e.g., with a replay attack or if the user accidentally submits it to the IC again). If a message has not shown up in the ingress history by the time it expires, the user knows it will never be executed and can create a new message to try again. In other systems this could lead to deadlocks or multiple unintended transfers.
- Since the expiry time of all ingress messages is at most a constant time interval of *max_expiry_interval* in the future, the time a message occupies space in the ingress history is bounded.
- There is no guarantee how much the subnet time deviates from the wall-clock time of the rest of the world, since finalization is only guaranteed in bursts of synchrony.
- In practice this does not pose a problem, as the user-experienced end-to-end latency for so-called *update calls*, i.e., calls that go through consensus and potentially change the state of some canister(s), is 1-4s and the current subnet time can be obtained together with a threshold signature of the subnet in less than 200ms.
- Having the subnet time in blocks and a notion of message expiry makes it possible to deduplicate ingress messages without having to keep the whole blockchain forever.

## 27.6 One Key to Rule Them All

To bring all the different parts together, we will now follow the full lifecycle of a message. When a user submits a valid request message $m$ for a canister to the IC, the networking layer broadcast it to the nodes of the subnet that hosts the canister. As a next step, it reaches the consensus layer, where the next blockmaker will include it in its block if the message signature and expiry check pass and there is still space in the block. If enough peers in the subnet validate the block successfully, then it will be notarized and eventually finalized. At this point, the finalized block's payload, including the message $m$ is passed on to the message routing layer.

As a first step, the current subnet time is set to the block's time stamp, $m$ is added to its target canister's ingress queue and an entry for $m$ with status `RECEIVED` is added to the ingress history. At some point $m$ will be scheduled to be processed by its canister. This may happen during the same execution round, or in one of the following execution rounds. If $m$'s expiry is below the current subnet time, then the ingress history entry for $m$ is set to `REJECTED`, otherwise it is set to `PROCESSING` and $m$'s execution starts. As part of the execution, messages to other canisters on the same or different subnets may be produced. These messages are added to the output queue of the canister. If the destination canister is on the same subnet, then the messages will be moved to the corresponding input queue next. For a destination canister on another subnet, all output queues for the same destination subnet are combined into a stream at the end of the execution round and the nodes create a threshold signature for the hash of this round's replicated state, which includes the ingress history, the streams and the time. Thus, the user can ask the IC for a message's status and obtain a signed response that can be trusted. Once a message for a canister on another subnet has made it into the other subnet's block and a certified reply has been generated and sent back after successful processing, then the original sender canister will execute the reply message and eventually complete $m$'s execution. At this point in time, the ingress history entry will be changed to contain the response and the status is set to `REPLIED`. Once the certification for this round has been created successfully, the user can learn the response.

In order to verify the threshold signature, the user needs to use the right public keys. To make this easy for the user, there is one IC master key that never changes. The nodes in the NNS subnet have shares of the corresponding private key. These shares are periodically rotated with a distributed protocol. When creating a new subnet, the NNS nodes generate key material for the new subnet nodes and create a certificate for the new subnet's public key. Subsequently, the certified state of each subnet is signed with this public key for each height. Hence, the complete verification process for the certified state of a given subnet consists of first verifying the NNS certificate for the subnet's public key followed by the verification of the signature of the replicated state. Since the creation of these threshold signatures requires $2f + 1$ out of $n$ signature shares, this guarantees that at least $f + 1$ honest nodes have participated in each state transition.

In summary, a single public key suffices to verify responses from the Internet Computer, in stark contrast to other blockchain platforms such as Bitcoin, where a full verification of *all* previous transactions is required to ensure that the considered transaction is valid.

## Chapter Notes

The IC's distributed nature and its replication are mostly abstracted away from the canister developers and users. Users can use their browsers to interact with canisters thanks to a translation mechanism on so-called gateway nodes.[3] The IC strives to provide latency and throughput similar to a traditional web application, to the extent possible for a globally distributed platform with strong security guarantees. Experiments and measurements are described on the IC wiki.[4]

The IC consensus protocol is described in more detail in [CDH+22] together with proofs for the communication complexity under several models with adapted protocol variants. With the current implementation and parametrization, around 2 blocks are produced per second. The IC dashboard[5] shows the block rate of each subnet as well as a myriad of other metrics.

To enable a protocol to be safe under asynchrony, all primitives must support this model, in particular also the (re)generation of threshold signature keys. [Gro21] describes how nodes can agree on BLS keys without requiring synchrony. This is an expensive process and is therefore executed only every 500 rounds for most subnets.

The canister execution and runtime environment of the IC is presented in [ABBK+23]. This paper describes the deterministic scheduling algorithm to pick the next message to be executed as well as the resource consumption accounting and memory subsystem, including experiments illustrating the performance of the system.

Developing a correct protocol and implementation for a complex system such as the IC is a difficult endeavor. Bugs sneak in easily both in the protocol design phase as well as in the implementation and during maintenance. [BDK+23] illustrates how model checking at runtime can be implemented to catch a variety of bugs.

This chapter was written in collaboration with Yvonne-Anne Pignolet and Thomas Locher.

## Bibliography

[ABBK+23] Maksym Arutyunyan, Andriy Berestovskyy, Adam Bratschi-Kaye, Ulan Degenbaev, Manu Drijvers, Islam El-Ashi, Stefan Kaestle, Roman Kashitsyn, Maciej Kot, Yvonne-Anne Pignolet, et al. Decentralized and stateful serverless computing on the internet computer blockchain. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, 2023.

[BDK+23] David Basin, Daniel Stefan Dietiker, Srđan Krstić, Yvonne-Anne Pignolet, Martin Raszyk, Joshua Schneider, and Arshavir Ter-Gabrielyan. Monitoring the internet computer. In *International Symposium on Formal Methods*, pages 383–402. Springer, 2023.

[3] https://internetcomputer.org/how-it-works/boundary-nodes/
[4] https://wiki.internetcomputer.org/wiki/Internet_Computer_performance
[5] See https://dashboard.internetcomputer.org/.

[CDH$^+$22] Jan Camenisch, Manu Drijvers, Timo Hanke, Yvonne-Anne Pignolet, Victor Shoup, and Dominic Williams. Internet computer consensus. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, pages 81–91, 2022.

[Gro21] Jens Groth. Non-interactive distributed key generation and key resharing. *Cryptology ePrint Archive*, 2021.