Chapter 26

Internet Computer

The *Internet Computer* (IC) realizes a world computer which 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 protocol mechanisms enabling this functionality.

26.1 Canisters and Subnets

Definition 26.1 (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 which 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 26.2 (Subnet)**.** *A* ***subnet*** *is a set of nodes providing state replication for the canisters deployed on it.*

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 of a subnet maintain a blockchain.

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

- 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 17.14.
- 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 and 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.

26.2 Networking

Definition 26.3 (IC networking layer)**.** *The* ***networking layer*** *of the IC delivers messages within a subnet,* ***reliably*** *and* ***efficiently****.*

Definition 26.4 (Communication complexity)**.** *The communication complexity of an algorithm is the number of bits that all correct nodes together send in the worst case.*

Remarks:

- Algorithm 18.11 has a communication complexity of $O(n^2L)$ for n nodes and a message of size L.
- Every node needs to receive the message, implying that at least $\Omega(nL)$ bits must be sent. Moreover, it can be shown that the communication complexity must be quadratic in n, regardless of the messages size, i.e., the lower bound is $\Omega(nL + n^2)$.
- It is further required that the nodes only deliver at most one message per broadcast.

Definition 26.5 (Integrity)**.** *A broadcast algorithm satisfies the* ***integrity property*** *if at most one message is delivered.*

Remarks:

- This requirement was not provided in Definition 18.10 and Algorithm 18.11.
- The following improvement fixes the integrity requirement.

Theorem 26.7. *Algorithm $\mathcal{A}_{bracha}$ implements reliable broadcast (with the integrity property) for $n > 3f$.*

Algorithm 26.6 Algorithm $\mathcal{A}_{bracha}$: Bracha's reliable broadcast

```
Source node $s$: Broadcast $m$
upon receiving $m$ from $s$ and not $sent\_echo$:
    Broadcast $(echo, m)$
    Set $sent\_echo$ to $true$
end upon
upon receiving $2f+1$ $(echo, m)$ or $f+1$ $(ready, m)$ messages:
    Broadcast $(ready, m)$
end upon
upon receiving $2f+1$ $(ready, m)$ messages:
    deliver($m$)
end upon
```

Proof. We first show that it is a reliable broadcast algorithm according to Definition 18.10. Assume that a correct node v delivers m, which means v must have received $2f+1$ $(ready, m)$ messages. Since there are at most f faulty nodes, at least $f+1$ correct nodes must have broadcast $(ready, m)$. Thus, all correct nodes eventually receive $(ready, m)$ at least $f+1$ times and they all broadcast $(ready, m)$. Consequently, they broadcast $(ready, m)$, causing all correct nodes to eventually receive this message at least $2f+1$ times and deliver it.

The integrity property follows from the observation that there can be only $2f+1$ echo messages for one message m because two messages m and m' with $2f+1$ echo messages each implies that there is at least one correct node that sent echo messages for m and m', a contradiction to the specification of the algorithm. If there are $2f+1$ $(echo, m)$, then there cannot be $f+1$ $(ready, m')$ messages for a different m' either because one of these messages must have come from a correct node. The first correct node to send $(ready, m')$ must have received $2f+1$ $(echo, m')$ messages. As argued before, this is not possible. □

Theorem 26.8. *Algorithm $\mathcal{A}_{bracha}$ has a communication complexity of $O(n^2L)$ in the asynchronous communication model.*

Proof. Every correct node other than the source node s broadcasts the message m of size L exactly once in an *echo* and a *ready* message. The node v_s further broadcasts m in the first step. The total communication complexity is therefore $nL + 2n^2L \in O(n^2L)$. □

Remarks:

- The algorithm also has the nice property that it terminates in constant time if every message arrives within constant time and the sender is correct.
- Since often $L \gg n$ in real systems such as the IC, the worst-case bandwidth consumption can be reduced considerably when using a more efficient algorithm.
- One approach is to broadcast a small data piece that identifies a message m uniquely with high probability first.

Definition 26.9 (Advert)**.** *An **advert** is a small, constant-size message used to advertise a larger message.*

Remarks:

- We use $\mathcal{H}(m)$ as the advert of a message m, where $\mathcal{H}$ is a cryptographically strong hash function (see Definition 18.25) that returns a hash of size H.
- After receiving an advert, a node can explicitly request the corresponding message.
- When receiving m for an advert ad, a node can then verify that m is correct, i.e., that $\mathcal{H}(m) = ad$.
- The following algorithm, $\mathcal{A}_{ad}$, is based on adverts in an attempt to keep the communication complexity low.

Algorithm 26.10 Algorithm $\mathcal{A}_{ad}$.

```
1: Execute $\mathcal{A}_{bracha}$ for message $advert := \mathcal{H}(m)$
2: upon receiving $m$ where $\mathcal{H}(m) = advert$ for the first time:
3:   Broadcast $\mathcal{H}(m)$
4:   deliver($m$)
5:   terminate after $(f+1)\Delta$ time
6: end upon
7: upon receiving $\mathcal{H}(m) = advert$ from $w$ for the first time:
8:   Store information about advertiser $w$
9: end upon
10: upon receiving $request(advert)$ from $w$:
11:   if message $m$ where $\mathcal{H}m) = advert$ is available and not sent to $w$ then
12:     Send $m$ to $w$
13:   end if
14: end upon
15: if received $advert$ and not sent $request(advert)$ in last $\Delta$ time then
16:   Chose some advertiser $w$ that was not chosen before
17:   Send $request(advert)$ to $w$
18: end if
```

Remarks:

- Algorithm $\mathcal{A}_{ad}$ first executes $\mathcal{A}_{bracha}$ to ensure that the nodes agrees on the advert for the message to be broadcast.
- Since $\mathcal{A}_{ad}$ has the notion of a timeout period Δ, it is expressed in the *synchronous* communication model.

Theorem 26.11. *For $\Delta := 2$, Algorithm $\mathcal{A}_{ad}$ implements reliable broadcast (with the integrity property) in the synchronous communication model.*

Proof. Assume that a correct node v delivers a message m and broadcasts $ad = \mathcal{H}(m)$ at time t. Every other correct node w receives this advert by time $t+1$

and requests m from a node that advertised m. After requesting m from faulty nodes for at most $f\Delta$ time, w requests m from either v or another correct node, which only advertises m if it has received m before. In either case, w receives m by time $t + 1 + (f + 1)\Delta$, proving the claim.

Algorithm $\mathcal{A}_{ad}$ satisfies the integrity property because $\mathcal{A}_{bracha}$ ensures that all correct nodes agree on the same advert $H(m)$ and, assuming that $\mathcal{H}$ is a collision-resistant hash function, the correct nodes may only deliver m. □

Theorem 26.12. *Algorithm $\mathcal{A}_{ad}$ has a communication complexity of $O(n^2H + (f + 1)nL)$ in the synchronous communication model.*

Proof. Executing Bracha's algorithm to reliably broadcast the advert costs $O(n^2H)$ bits. Since every correct node broadcasts an advert only once, the communication complexity for adverts is $O(n^2H)$. This term encompasses the communication complexity for requests.

Every correct node only receives the message m from one other correct node, requiring $O(nL)$ bits. However, every malicious node may request the message m from each correct node, i.e., the communication complexity for message transfers is $O((f + 1)nL)$. □

Remarks:

- If $f \in O(1)$ and $nH \in O(L)$, we get an asymptotically optimal communication complexity of $\Theta(nL)$; however, the algorithm runs for $O(n\Delta)$ time, which is significantly worse than $\mathcal{A}_{bracha}$,
- We only get the improved communication complexity in the synchronous model and only if there are few faulty nodes.
- In the asynchronous communication model, the algorithm does not guarantee the properties of reliable broadcast. For example, the first property is violated when a correct node delivers m and then terminates after $(f + 1)\Delta$ time but before the other nodes can send their requests for message m. In this model, the algorithm is only correct if it *never terminates*. However, even in this case, the communication complexity is $O(n^2L)$.
- A *coding-based* alternative does not have these shortcomings.

Definition 26.13 (Erasure code)**.** *A (n, k)-**erasure code** is a code that transforms a message of k **symbols** (of some given alphabet) into a message with $n > k$ symbols such that the original message can be recovered from a subset of the n symbols.*

Remarks:

- We consider *optimal* erasure codes, which have the property that any subset of k symbols is sufficient to reconstruct the original message.
- When using an optimal erasure code, it is possible to split a message of size $L \geq k$ into n *fragments* of approximate size L/k each (consisting of one or more symbols). A fragment may be slightly larger, e.g., padding the original message to a size that is evenly divisible by k.

- The basic idea is to set $k := f+1$ and encode n fragments of size $L/(f+1)$, n being the number of nodes in the subnet. The sender can send each node a different fragment, which the receiving nodes broadcast. Each node can then reconstruct the message m when receiving $f+1$ different and *valid* fragments.
- Note that at least $f+1$ fragments must be required, otherwise the f malicious nodes could fabricate any message themselves.
- Each fragment must be a *valid* encoding. How can we ensure that malicious nodes do not send random data instead of valid fragments?

Definition 26.14 (Merkle tree)**.** *A **Merkle tree** is a tree in which every leaf is labeled with the hash of a data block, and every inner node is labeled with the hash of the labels of its children.*

Remarks:

- The n fragments are placed at the leaves. For example, if there are 4 fragments $f_1, \ldots, f_4$, we would get the Merkle tree depicted in Figure 26.15, where $h_1 := \mathcal{H}(\mathcal{H}(f_1)|\mathcal{H}(f_2))$, $h_2 := \mathcal{H}(\mathcal{H}(f_3)|\mathcal{H}(f_4))$, and $h_0 := \mathcal{H}(h_1|h_2)$. The operator $|$ denotes the concatenation of the two given hash values.
- A proof consists of all the hashes on the path from the root to the fragment plus the hash of the fragment with the same parent node. It follows that the number of hashes is logarithmic in the number of fragments.

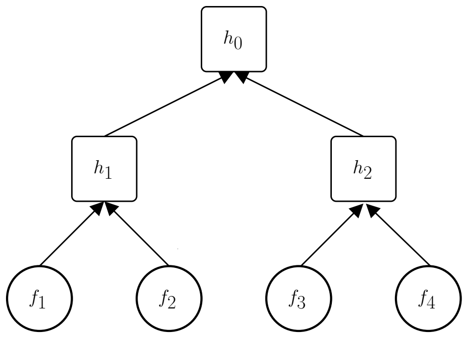

Figure 26.15: A Merkle tree for 4 fragments $f_1, \ldots, f_4$ is shown. It holds that $h_i := \mathcal{H}(\mathcal{H}(f_{2i-1})|\mathcal{H}(f_{2i}))$ for $i = 1, 2$ and $h_0 := \mathcal{H}(h_1|h_2)$.

Remarks:

- The algorithm $\mathcal{A}_{code}$ also uses $\mathcal{A}_{bracha}$ to first agree on the message identifier. Unlike algorithm $\mathcal{A}_{ad}$, algorithm $\mathcal{A}_{code}$ uses the Merkle root hash h_0 as the message identifier, which is broadcast reliably.
- The validity of received fragment is verified against the reliably broadcast root hash h_0.

Algorithm 26.16 Algorithm $\mathcal{A}_{code}$ at node v_i. Initially, $root\_hash = \perp$, $F = \{\}$, and $m = \perp$.

```
(f_1, ..., f_n) := get_fragments(m)
h_0 := get_merkle_root_hash((f_1, ..., f_n))
Execute A_bracha for message root_hash := h_0
for j ∈ {1, ..., n} do
  P_j := get_merkle_proof((f_1, ..., f_n), j)
  Send (f_j, P_j, j) to v_j
end for

upon receiving (f, P, j) and root_hash ≠ ⊥:
  if valid(f, P, root_hash, j) then
    F := F ∪ {f}
    if i = j and not broadcast f before then
      Broadcast (f, P, i)
    end if
  end if
end upon

if |F| = f + 1 and m = ⊥ then
  m := recover_message(F)
  (f_1, ..., f_n) := get_fragments(m)
  h_0 := get_merkle_root_hash((f_1, ..., f_n))
  if h_0 = root_hash then
    if not broadcast f_i before then
      P_i := get_merkle_path((f_1, ..., f_n), i)
      Broadcast (f_i, P_i, i)
    end if
  else
    root_hash := ⊥
  end if
end if

if |F| = n − f and m ≠ ⊥ and root_hash ≠ ⊥ and not delivered then
  deliver(m)
end if
```

- If the recovered message is not consistent with the broadcast root hash, it is save not to deliver anything because all correct nodes that recover the message must reach the same conclusion.

Theorem 26.17. *Algorithm $\mathcal{A}_{code}$ implements reliable broadcast (with the integrity property) in the asynchronous communication model.*

Proof. Let v be a correct node that delivers m. Since $n - f > n - 2f = f + 1$, $m \neq \perp$, and $root\_hash \neq \perp$, v must have successfully reconstructed the message and all fragments beforehand and broadcast its fragment. Since $|F| = n - f$, v must have received fragments from at least $n - 2f$ correct nodes. These $n - 2f$ correct nodes have broadcast their fragments, which implies that all correct nodes eventually receive at least $n - 2f = f + 1$ fragments. According to the algorithm, all correct nodes then reconstruct the message m successfully (because v reconstructed it successfully) and broadcast their fragments as well if they have not already done so earlier. Hence it follows that all correct nodes eventually broadcast their fragments and, since there are at least $n - f$ correct nodes, $|F| = n - f$, $root\_hash \neq \perp$, and $m \neq \perp$ holds eventually, causing all correct nodes to deliver the message m.

The integrity property holds because only one message is delivered. □

Theorem 26.18. *Algorithm $\mathcal{A}_{code}$ has a communication complexity of $O(n^2 \log(n)H + nL)$ in the asynchronous communication model.*

Proof. The initial reliable broadcast has a communication complexity of $O(n^2H)$. For a message of size L, the fragment size is $L/(f + 1) \approx 3L/n$. A Merkle proof consists of approximately $\log(n)$ hashes of size H each. The n messages from the sender thus require $n \cdot (3L/n + \log(n)H) \in O(L + n\log(n)H)$ bits to be sent. Subsequently, every node broadcasts its fragment, together with the Merkle proof, at most once, which requires at most $n^2 \cdot (3L/n + \log(n)H) \in O(nL + n^2 \log(n)H)$ bits. □

Remarks:

- The communication complexity is asymptotically optimal for $L \in \Omega(n \log(n)H)$.
- If the sender is correct and messages arrive within constant time, the algorithm also terminates in a constant time.
- While algorithm $\mathcal{A}_{code}$ is superior to $\mathcal{A}_{ad}$ in theory, the IC currently implements a variant of $\mathcal{A}_{ad}$ because the communication complexity of algorithm $\mathcal{A}_{ad}$ is effectively nL in practice, ignoring the small adverts. The bandwidth consumption only increases in adverse network conditions or when nodes are faulty, which fortunately happens rarely. Moreover, as we will learn later, messages *expire* after a certain time, bounding the time during which messages must be retained so that they can be delivered to requesting nodes. By contrast, algorithm $\mathcal{A}_{code}$ sends roughly $3nL$ bits when everything works as desired (all n nodes broadcast their fragments of size $3L/n$). As a result, the bandwidth consumption would be considerably higher in the steady state.

- There are coding-based reliable algorithms that are more efficient. Changing requirements may yet necessitate the transition to such a coding-based approach.

26.3 Consensus

Definition 26.19 (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 26.20 (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.
- Multi-signatures comprising $n - f$ signatures are used. A public key can be used to verify multi-signatures issues by any subset of the nodes of size at least $n - f$.

- Additionally, the algorithm uses a *BLS threshold signature scheme* (see Definition 18.19) 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 26.21 (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).

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.

Definition 26.23 (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.

Definition 26.24 (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 plus* 1.

Remarks:

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

Definition 26.25 (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.*

Algorithm 26.22 IC Consensus: Actions at node v_i for epoch h

```
1: $r_i$ := rank($i$, beacon($h$))
2: if at least $2r_i + \varepsilon$ 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
```

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.

- A node with rank $r \in \{0, \ldots, n-1\}$ 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, v_i 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 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".

- If this is the only block for which v_i 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]

[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.

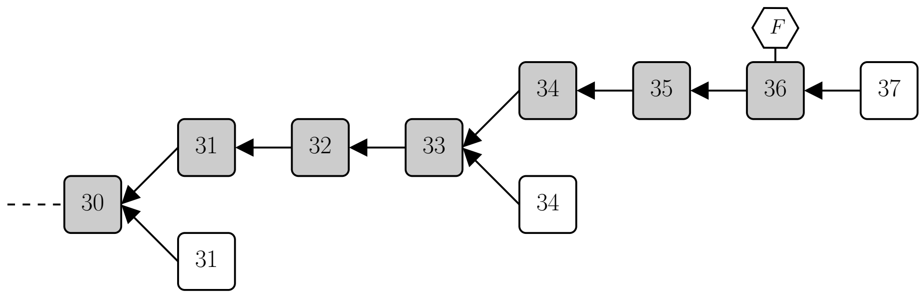

Figure 26.26: A possible block tree with forks in epochs 31 and 34. Since a block was finalized in epoch 36, all its predecessors are implicitly finalized as well.

Remarks:

- Figure 26.26 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 26.27. *For every epoch h, if correct nodes v and v' finalize blocks B and B', then $B = B'$.*

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

Lemma 26.28. *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 beforehand. 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. $\square$

Lemma 26.29. *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 26.28, 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 26.30. *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. □

26.4 Message Routing and Execution Environment

Definition 26.31 (IC message routing layer)**.** *The **message routing layer** of the IC ensures the messages from consensus reach their destination canister and are enqueued for processing, persists state changes and communicates with other subnets and provides authenticated information to the users.*

Definition 26.32 (IC execution environment layer)**.** *The **execution environment layer** of the IC schedules and processes canister messages.*

Definition 26.33 (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 such 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.

Definition 26.34 (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 canisters communicate with.*

Remarks:

- C' and C'' may reside on the same or on different subnets.
- The message routing layer inserts messages from blocks in one of multiple input queues.
- For each block height, the execution layer will consume some of the messages in the input queues and update the replicated state of the relevant canisters and create messages in the output queues.
- Subsequently, 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.*
- The whole message routing process taking place for each finalized block is visualized in Figure 26.35.

Definition 26.36 (Ingress message status)**.** *An ingress message from a user 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** of a node.*

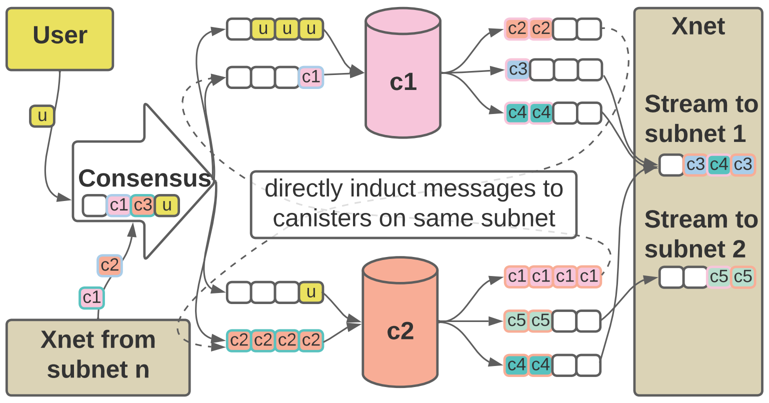

Figure 26.35: Routing messages through the IC protocol stack. Messages for canisters, issued by users or canisters on other subnets, are validated and ordered by consensus. Subsequently, messages are put into the input queues of their destination canisters. Messages created by canisters are put into output queues from where they are either transferred to their respective input queues on the same canister (bypassing consensus) or sent as part of streams to their target subnet.

Remarks:

- Users can query the ingress history of the IC to learn about the status of their message.
- A message transitions from one status to the next until it is `REPLIED` or transitions to `REJECTED` from any earlier status directly.

Definition 26.37 (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 and streams, the ingress history and the current IC 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, 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 taken in practice.
- From a user's point of view, the IC time is the timestamp associated with the latest certified height.

- The node does not wait for all certification shares to arrive but continues with xnet communication 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 and other subnets rely on signatures constructed from certificates on ingress history and xnet messages, respectively. Xnet messages and responses are Merkle tree leaves. The signature of a Merkle tree leaf 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". I.e., if the difference between the notarized height and the certified height exceeds a threshold, then the time how long 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 26.38 (Block payload)**.** *A block payload comprises*

- *time: timestamp of block creation*
- *ingress_payload: set of messages from users to canisters on this subnet*
- *xnet_payload: set of messages from canisters on other subnets to canisters on this subnet*
- *$certified\_height$: state certificate for this height exists*

Definition 26.39 (Extended Block Validity)**.** *A node considers a block valid if*

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

- *none of the ingress and xnet 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 and no message from xnet streams is skipped.*

Remarks:

- To create a valid block B in `build_block()`, the consensus layer of a node can
  - Set $B.time$ to its current local system time.
  - Select a set of ingress messages that have not occurred in any of the predecessor blocks for which the expiry time is in $(B.time, B.time + max\_expiry\_interval)$ for $B.ingress\_payload$.
  - For $certified\_height$ and $xnet\_payload$, the consensus layer calls message routing's `MR.create_payload` function to obtain the latest height for which the nodes has a replicated system state certification and a set of valid messages from other subnets. To this end, consensus provides all xnet messages in blocks since the last finalized block as input, to ensure that only new xnet messages are in the created payload.
- To decide if a block from another node `is_valid()`, the consensus layer can check the first five validity conditions without the support of the message routing layer using its current local system time, the expiry time of the ingress messages in the block, and the locally available predecessor blocks. To verify the message routing parts of the block, consensus can use `MR.is_payload_valid`, which checks if the certified height is not above the height for which the node has a valid full certified state signature, all xnet messages in B have correctly signed Merkle tree witnesses, and no messages from xnet streams are skipped. To this end, the set of previous xnet messages provided by Consensus must contain all xnet messages in blocks since $B.certified\_height$ in *tree*.
- If the local system time of the nodes diverges too much, then no blocks will be notarized.
- Blocks below $B.certified\_height$ and expired ingress messages are no longer needed for block validity and can be discarded.

Algorithm 26.40 Message Routing: Triggered actions by the consensus layer

```
upon process_payload($time, ingress, xnet$) called:
  $state.height := state.height + 1$
  $state.time := time$
  insert_into_input_queues($ingress \cup xnet, state$)
  for $m \in B.ingress$ do
    $state.ingress\_history[m]$ := RECEIVED
  end for
  while execution time left do
    $m$ := pop_from_input_queues($state$)
    if $m \in state.ingress\_history$ then
      if $state.time < m.expiry$ then
        $state.ingress\_history[m]$ := PROCESSING
      else
        $state.ingress\_history[m]$ := REJECTED
      end if
    end if
    execute($m, state$)
  end while
  prune_ingress_history($state$)
  certify_state($state$)    //sets $certified\_height$ eventually
  xnet_communication()
end upon

upon create_payload($previous\_xnet$) called:
  // pick xnet messages disjoint from $previous\_xnet$
  $xnet\_payload$ := select_xnet_messages($previous\_xnet$)
  return ($xnet\_payload, certified\_height$)
end upon

upon is_payload_valid($B$, $previous\_xnet$) called:
  // check message routing parts of block
  return  $B.certified\_height \leq certified\_height$ and
   xnet_valid($B.certified\_height, B.xnet\_payload, previous\_xnet$)
end upon
```

Remarks:

- All message routing steps in `process_payload()` and `is_payload_valid()` must be fully deterministic.
- Messages for different canisters can be executed in parallel.
- The execution of a message may fail. For ingress messages, the ingress history is updated to status `REJECTED` in this case. If the message execution creates new messages to the same or other canisters and needs to wait on their response, the status remains `PROCESSING`. If a response to the message has been generated, the ingress history status is set to `REPLIED`, allowing the user to collect the response.

- `prune_ingress_history` removes every message m from the ingress history if its 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 triggered by `certify_state`. When the full signature of a state height $> certified\_height$ 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.

- 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. When `xnet_communication()` is called, 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 subnets using the function `xnet_signatures_valid`.

- `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.

- 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.

Theorem 26.41 (Unique execution before expiry)**.** *Every ingress message will either enter the state `PROCESSING` in Algorithm 26.40 exactly once before its expiry time with respect to the IC time or it will never be processed. Subnet-signed IC responses guarantee that IC time is strictly monotonic and that the reported ingress history status transitions have occurred at $f + 1$ nodes or more.*

Proof. Correct nodes adhere to the block validity conditions and only create and notarize blocks with a higher timestamp than previous blocks. Any finalized block has been notarized by $n - f$ nodes, hence it holds for any subnet with at most f byzantine nodes that the IC 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 IC 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 accidentally be executed twice (e.g., with a replay attack or if the user 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 IC 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 IC time can be obtained together with a threshold signature of the subnet in less than 200ms.

- Having the IC time in blocks and a notion of message expiry makes it possible to deduplicate ingress messages without having to keep the whole blockchain forever.

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 boundary 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

[3]`https://internetcomputer.org/how-it-works/boundary-nodes/`

[4]`https://wiki.internetcomputer.org/wiki/Internet_Computer_performance`

adapted protocol variants. With the current implementation and parametrization, around 1 block is produced per second in most subnets. 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.

[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.

[5]See `https://dashboard.internetcomputer.org/`.