Lean Consensus

Contents

1.0 Introduction: why consensus exists

Ethereum [1] is a distributed system: many independent computers (called nodes) communicate over a network that can be slow, unreliable, or adversarial. The purpose of a blockchain protocol is to let these nodes maintain a shared view of an evolving system state, even when some participants fail or misbehave.

A convenient mental model is a replicated state machine. A state machine is a function that takes (i) a current state and (ii) an input, and produces a new state. In Ethereum, the inputs are transactions (and related protocol messages), and the state contains things like account balances, smart contract code, and contract storage. The word replicated means that every honest node attempts to compute the same sequence of state transitions, so that they converge to the same result.

This raises an immediate challenge: in an open peer-to-peer network, different nodes may see different messages first, or may be targeted by conflicting information. Consensus is the part of the protocol whose job is to make the network agree on a single history of blocks. Once there is agreement on the block history, the order of transactions is fixed, and the state transitions become unambiguous.

1.1.0 Blocks, chaining, and local verification

The core data structure of a blockchain is a sequence of blocks. Each block bundles transactions together with metadata, and contains a cryptographic reference (a hash) to its parent block. This creates a chain: if any past block is modified, its hash changes, which breaks the link to all later blocks. Figure 1.1 illustrates this hash-linked structure.

Ethereum nodes do not accept blocks on trust. Instead, they verify blocks locally by re-executing the included transactions (and the protocol rules around them). This redundant verification is essential: it prevents a malicious peer from convincing you to accept an invalid state transition.

It is often useful to separate the protocol into two roles:

- A set of rules that define what it means for a block to be *valid* (the state transition function, plus any validity conditions).
- A set of rules that define which valid blocks are treated as *canonical* when there are competing histories (fork choice and finality).

In Ethereum today, these roles are commonly described as the execution layer and the consensus layer respectively. In this document we focus on the consensus side of the story, and in particular on a proposed redesign commonly referred to as Lean Consensus.

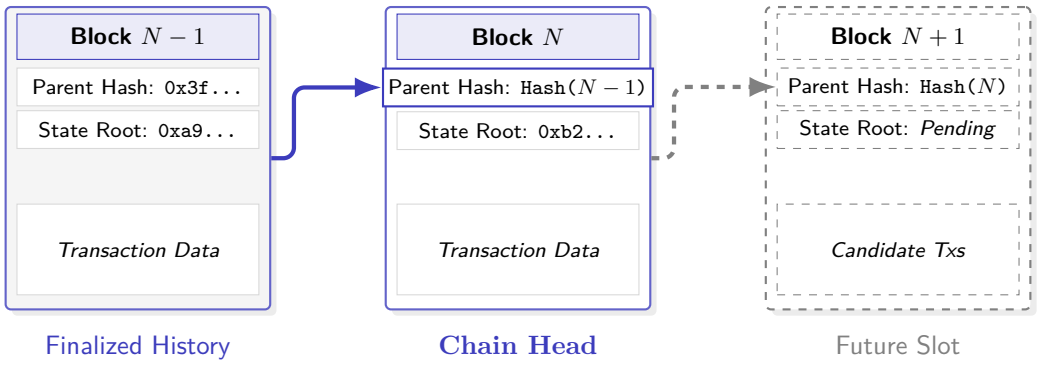

Figure 1.1: The cryptographic chain structure. Notice how the *Parent Hash* field of Block N explicitly commits to the entire content of Block $N-1$. Changing any byte in Block $N-1$ would change its hash, breaking the link to Block N.

1.2.0 What a consensus mechanism must guarantee

In a distributed network, disagreement is not an exceptional event but a normal condition. Nodes do not receive messages at the same time, network links may be slow or unreliable, and different participants may temporarily observe different versions of the system state. If left unchecked, these differences would cause the network to fragment into incompatible histories.

A consensus mechanism exists to prevent this fragmentation from becoming permanent. Its role is not to eliminate disagreement entirely, but to ensure that honest nodes repeatedly reconverge on a single shared history of the blockchain.

1.2.1 Block creation and fork resolution

To understand what this requires, it is helpful to separate the problem into two recurring situations that arise during normal operation.

1. *Block creation.* At any given moment, more than one participant may be capable of proposing a valid next block. The protocol must therefore specify who is allowed to propose blocks, when they may do so, and under which constraints. Without such rules, competing proposals would quickly overwhelm the system.

2. *Fork resolution.* Because messages propagate with delay, different nodes may temporarily build on different blocks. A consensus mechanism must specify how nodes choose between these competing histories so that the network eventually reconverges on a single chain. This process is illustrated in Figure 1.2.

1.2.2 Long-term guarantees

Because a blockchain is intended to operate continuously, these mechanisms must satisfy long-term guarantees that hold over the lifetime of the system, not just in individual cases.

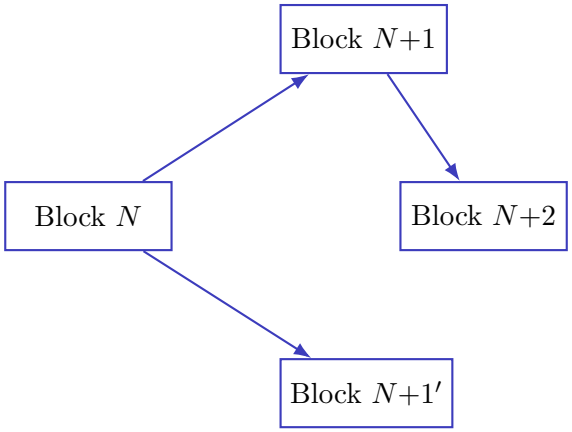

Figure 1.2: Temporary forks caused by network delays are resolved by the consensus rules, allowing the network to reconverge on a single chain.

1. *Safety.* Informally, safety means that the protocol should not lead honest nodes into an irreversible contradiction. Once part of the chain is considered final, honest nodes should not later finalize a conflicting history.

2. *Liveness.* Informally, liveness means that the system continues to make progress. Even in the presence of delays and partial failures, new valid blocks should keep being produced, and the chain should eventually advance.

There is one further requirement that is specific to public blockchains. Since anyone can join the network, the protocol must defend itself against Sybil attacks, where an adversary creates many fake identities to gain disproportionate influence. Consensus mechanisms address this by tying participation to a scarce resource. In Ethereum's history, this resource was first external computation in Proof of Work, and later economic collateral in Proof of Stake.

1.2.3 From Proof of Work to Proof of Stake

In Proof of Work (PoW), the ability to propose blocks is tied to expending external resources (hardware and electricity). In Proof of Stake (PoS), it is tied to posting collateral inside the protocol itself.

Under PoS, the main actors are validators. A validator is a protocol participant that locks up ETH as stake and then runs software that proposes blocks and votes on blocks. If a validator violates key rules (for example, by equivocating—signing conflicting messages), the protocol can penalize them by slashing a portion of their stake. Figure 1.3 summarizes the shift in the resource that secures the system.

1.3.0 Lean Consensus as a redesign effort

Ethereum's current PoS consensus layer is specified by the Beacon Chain design. Since that design was created, both the research landscape and the operational realities of Ethereum have evolved. A recurring motivation for Lean Consensus is to treat the consensus layer as something

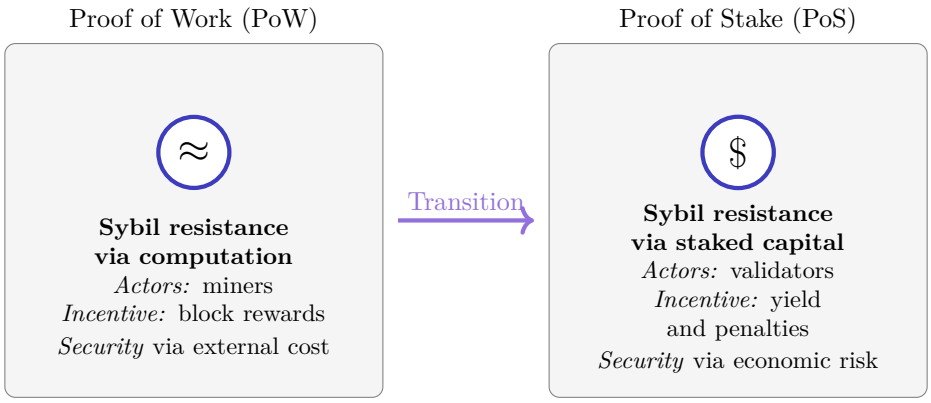

Figure 1.3: PoW and PoS use different Sybil-resistance resources. PoW relies on external expenditure; PoS relies on collateral that can be penalized by the protocol.

that can be simplified and modernized using lessons learned from the research front and real-world deployment.

Lean Consensus is commonly described as a holistic redesign. The intent is to revisit core components (such as validator responsibilities, committee structures, signatures, block production flow, etc) with a bias toward minimality, clearer invariants, and better alignment with the surrounding ecosystem.

1.3.1 Ossification accelerationism

Large protocol changes have coordination costs: they require client implementations, testing, audits, and social alignment. One proposed deployment philosophy for Lean Consensus is to batch major architectural changes into one big upgrade, instead of spreading them across many incremental hard forks.

The word ossification refers to reaching a stable maintenance mode where the core protocol changes rarely and only for well-understood reasons (such as security fixes). Ossification accelerationism is the argument that, if a major redesign is desirable, it can be better to do it deliberately in one concentrated effort, rather than living for many years with an increasingly complex intermediate design.

1.4.0 How to read this document

This text is written as an educational companion to the executable specifications in `leanSpec`. The goal is to make each concept self-contained before it is used. When new terminology is introduced, it will be defined and then reused consistently.

We proceed bottom-up: starting from the core abstractions (time, blocks, votes, finality, etc), then building toward the full Lean Consensus architecture and the rationale behind each design choice.

2.0 Simple Serialize (SSZ)

Consensus protocols must solve a pretty important problem: how can thousands of independent nodes verify that they share the same view of the world? When two validators want to compare their copies of the state, they cannot afford to transmit millions of bytes over the network. What they need is a compact, deterministic fingerprint of their data—one that changes if even a single bit differs.

This chapter introduces Simple Serialize, or SSZ, the serialization and Merkleization scheme that underpins Lean Consensus. SSZ serves three distinct purposes in the protocol:

1. **Consensus.** Nodes must agree on a canonical representation of protocol objects. If two nodes hold identical data, their serializations must be byte-for-byte identical. This property allows validators to compute cryptographic hashes of their states and compare short digests instead of entire data structures.

2. **Communication.** When blocks and attestations propagate through the peer-to-peer network, they must be serialized for transmission. The receiving node must reconstruct identical objects from the byte stream.

3. **Proofs.** Light clients cannot store the full beacon state. Instead, they rely on Merkle proofs to verify specific pieces of data against a trusted root hash. SSZ's Merkleization scheme makes such proofs compact and efficient.

Unlike self-describing formats such as JSON or Protocol Buffers, SSZ requires both sender and receiver to know the schema in advance. This design choice keeps the encoding minimal—there are no field names or type tags embedded in the serialized data. The tradeoff is that a raw SSZ blob is meaningless without knowledge of the type it represents.

2.1.0 Design Philosophy: Why SSZ?

Before detailing the mechanics, it is useful to understand why Ethereum migrated from the Recursive Length Prefix (RLP) serialization used in the execution layer to Simple Serialize (SSZ) for the consensus layer. The design of SSZ is driven by three specific requirements of a Proof-of-Stake protocol:

1. **Random Access.** In RLP, data is packed linearly. To read the last field of an object, a node must parse and decode every preceding field. This is acceptable for small transactions but prohibitive for the State, which contains hundreds of thousands of validators.

SSZ utilizes a schema-based offset approach. Because the shape of the data is known in advance, a node can jump directly to a specific validator's record without parsing the gigabytes of data preceding it.

2. **Unique Representation (Bijectivity).** Consensus requires that for any valid object, there is exactly one valid byte sequence. If two different byte streams could deserialize into the same object, malicious actors could split the network by broadcasting mathematically identical blocks with different hashes. SSZ enforces a strict one-to-one mapping between value and serialization.

3. **Merkle-Native Structure.** As we will see in later sections, SSZ is designed such that the serialization structure maps cleanly to Merkle tree leaves. This allows for the generation of compact multiproofs, enabling light clients to verify specific state fields (like a single balance) without downloading the full state.

2.2.0 The Type System

SSZ operates on a strict, pre-defined schema. Every piece of data in the consensus layer belongs to one of two categories: Basic Types, which represent fundamental values, or Composite Types, which construct complex data structures from other types. The relationship between these types is illustrated in Figure 2.1.

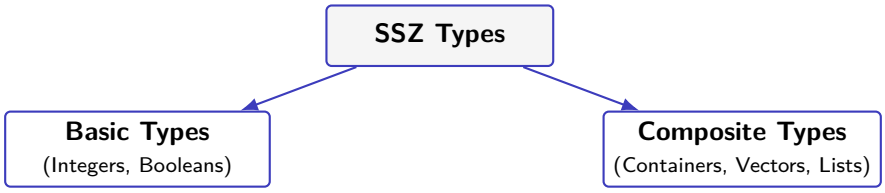

Figure 2.1: The SSZ type hierarchy distinguishes between fundamental values and structured data.

2.2.1 Basic Types

Basic types have a fixed size and are serialized directly.

- **Unsigned Integers (`uintN`):** SSZ supports unsigned integers of fixed bit-lengths: $N \in \{8, 16, 32, 64, 128, 256\}$. Crucially, serialization uses little-endian encoding. For instance, the value 1 represented as a `uint32` is serialized as the byte sequence `0x01000000`, placing the least significant byte first.

- **Booleans:** A logical value stored as a single byte: `0x01` for `True` and `0x00` for `False`.

2.2.2 Composite Types

Composite types combine multiple elements into a single structure. The serialization behavior of a composite type depends heavily on whether its size is fixed or variable. Table 2.1 details the composite structures used in the protocol.

Type	Notation	Fixed Length	Definition
Vector	`Vector[T, N]`	✓	A fixed-length sequence containing exactly N elements of type T.
List	`List[T, N]`		A variable-length sequence containing *up to* N elements of type T.
Container	`class Name`	–	An ordered collection of named fields. It is considered *variable-size* if it holds any variable-size fields.
Bitvector	`Bitvector[N]`	✓	A fixed-size array of N bits, packed densely (8 bits per byte).
Bitlist	`Bitlist[N]`		A variable-length array of up to N bits, packed densely.
Union	`Union[T1, T2...]`		A wrapper holding exactly one value from a set of possible types. It is serialized with a single selector byte followed by the value.

Table 2.1: Composite Types in SSZ. The column Fixed Length indicates if the type enforces a static element count (N) or a dynamic capacity (up to N).

2.2.3 Default Values and Zero-ness

In Ethereum, empty states are strictly defined. When a new validator record is created or a field is cleared, it must revert to a recursive default value. This determinism is essential for calculating the zero state of the protocol.

- **Basic Types:** `uintN` defaults to 0. `boolean` defaults to `False`.
- **Vectors/Bitvectors:** Default to a sequence of N default values.
- **Lists/Bitlists:** Default to an empty sequence (length 0).
- **Containers:** Default to a container where every field is set to its recursive default.

An object is said to be *zeroed* if it matches this recursive default structure perfectly.

2.3.0 Serialization Mechanics

SSZ employs a split-layout strategy to handle complex data structures efficiently. Unlike simple concatenation, which forces a deserializer to read every byte sequentially to find a specific field, SSZ aims to preserve random access properties even when variable-length data is present.

To achieve this, the serialized output of a container is partitioned into two distinct regions:

1. **The Fixed Part.** This section stores all fixed-size values directly. For variable-size values, it stores a 4-byte offset (a pointer) instead of the actual data.

2. **The Variable Part.** This section resides at the end of the byte stream and contains the actual raw data for all variable-size fields, packed sequentially.

The offset represents the number of bytes from the start of the serialized object to the start of the specific variable-length data.

2.3.1 Visualizing Offsets

Consider a hypothetical `ValidatorRecord` container with three fields:

1. `id` (type `uint16`, fixed 2 bytes)
2. `signatures` (type `List`, variable length)
3. `pubkey` (type `bytes48`, fixed 48 bytes)

In a naive concatenation scheme, reading the `pubkey` would require parsing the entire `signatures` list first. In SSZ, the `pubkey` remains at a fixed position because the list is replaced by a fixed-size offset. This mechanism is illustrated in Figure 2.2.

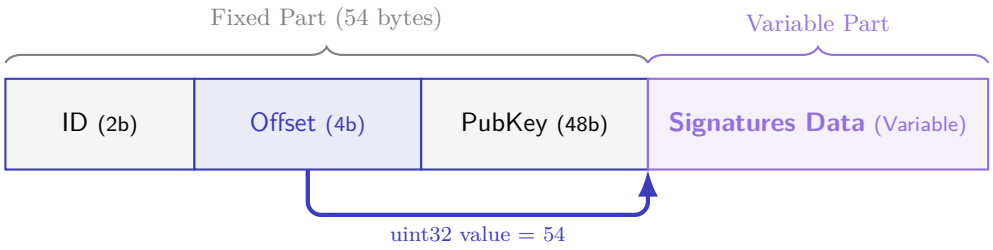

Figure 2.2: The SSZ Offset Scheme. Fixed-length data allows consistent memory strides, while variable-length data is pushed to a heap at the end.

2.3.2 Bitlists and the Sentinel Bit

The `Bitlist` type presents a unique serialization challenge regarding precision. Since computers address memory in bytes (8 bits), but a bitlist may contain an arbitrary number of bits (e.g., 5 bits), ambiguity arises.

If we serialize the bit sequence `10101` into a byte, we might get `00010101`. However, upon deserialization, it is impossible to know if the original data was 5 bits (`10101`) or 8 bits (`00010101`).

SSZ resolves this by appending a sentinel bit. A single `1` bit is added immediately after the significant data bits. During deserialization, the system reads the byte, locates the most significant `1` bit, and identifies it as the sentinel. All bits less significant than the sentinel are data; the sentinel and all bits more significant are discarded. Figure 2.3 demonstrates this packing strategy.

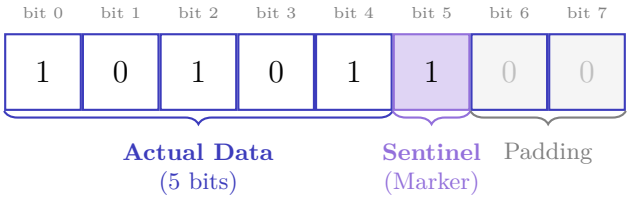

Figure 2.3: Bitlist Serialization. To preserve the exact length of the bit sequence within byte boundaries, a `1` (sentinel) is appended immediately after the data. The deserializer finds the highest-index `1` to determine the list length.

2.3.3 Deserialization Hardening and Security

Because SSZ is an external input, deserializers must not naively follow the offsets found in the byte stream. Malicious actors could craft invalid SSZ data to trigger buffer overflows or memory exhaustion. A secure deserializer must enforce strict validation checks. Figure 2.4 illustrates the three primary classes of invalid serialization that must be rejected.

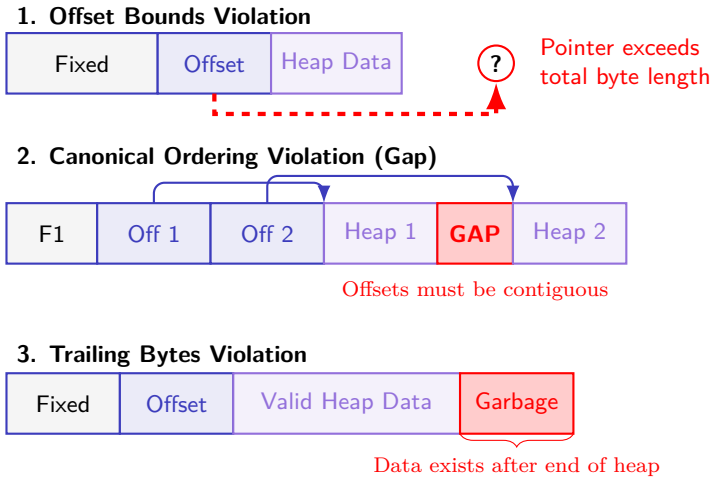

Figure 2.4: Security Checks. A valid SSZ deserializer must reject payloads where offsets point out of bounds (1), leave gaps between variables (2), or contain unused garbage data at the end (3).

- **Offset Bounds:** Every offset must point to a valid index within the received byte buffer. An offset pointing beyond the length of the data implies a missing payload or a malformed stream.
- **Canonical Ordering:** Offsets must be strictly increasing and contiguous. The first variable-length element must begin immediately after the fixed part, and the start of element $N+1$ must equal the end of element N. Gaps (unused bytes between heap elements) are forbidden.
- **No Trailing Bytes:** The end of the last variable-length element must align exactly with the end of the byte stream. Any extra bytes remaining after the last field indicate a non-canonical serialization.

2.3.4 Protocol Integration

SSZ serves as the format for both consensus and API communication. However, two practical adaptations are made for these contexts:

1. **JSON Mapping.** Since JSON does not natively support 64-bit integers or raw bytes, SSZ types map to specific JSON representations. `uint64` and larger are represented as strings (e.g., `"1000"`) to preserve precision in JavaScript environments. Byte arrays are represented as hex-strings (e.g., `"0x1234"`).
2. **Network Compression.** While SSZ structure is sparse (containing many zero-padded fields), it is rarely transmitted raw over the peer-to-peer network. The transport layer wraps SSZ payloads in Snappy compression, significantly reducing the bandwidth requirement.

2.4.0 Merkleization and Hash Tree Roots

While serialization converts objects into linear byte streams for network transmission, Merkleization converts objects into a 32-byte digest known as the Hash Tree Root. This root serves as the unique cryptographic identity of the object.

SSZ is designed to be Merkle-native, meaning the structure of the data directly dictates the topology of its Merkle tree. This design enables the creation of efficient proofs, allowing light clients to verify individual fields of a massive state object without downloading the entire dataset. The process occurs in three distinct stages: packing, tree construction, and length mixing.

2.4.1 Chunks and Packing

The fundamental unit of the SSZ Merkleization process is the 32-byte chunk. Before a data structure can be hashed, it must be mapped onto a standardized grid of these chunks.

- **Basic Types.** Small primitive types are packed tightly into chunks to maximize efficiency. For instance, a `uint64` occupies 8 bytes. Therefore, four `uint64` values fit perfectly into a single 32-byte chunk.
- **Composite Types.** Complex objects, such as Containers, are not packed directly. Instead, they are represented by the Hash Tree Root of their contents.

Figure 2.5 illustrates this packing efficiency. In this example, a list of six 64-bit integers is packed into two chunks, rather than occupying six separate leaves.

2.4.2 Tree Construction

Once the data is organized into a list of chunks, SSZ builds a binary Merkle tree. If the number of chunks is not a power of two, the list is implicitly padded with zero-chunks until it reaches the next power of two. The tree is hashed upward using SHA-256 until a single root remains.

This power-of-two requirement is crucial for the generation of generalized indices, as it ensures the tree has a predictable, balanced depth.

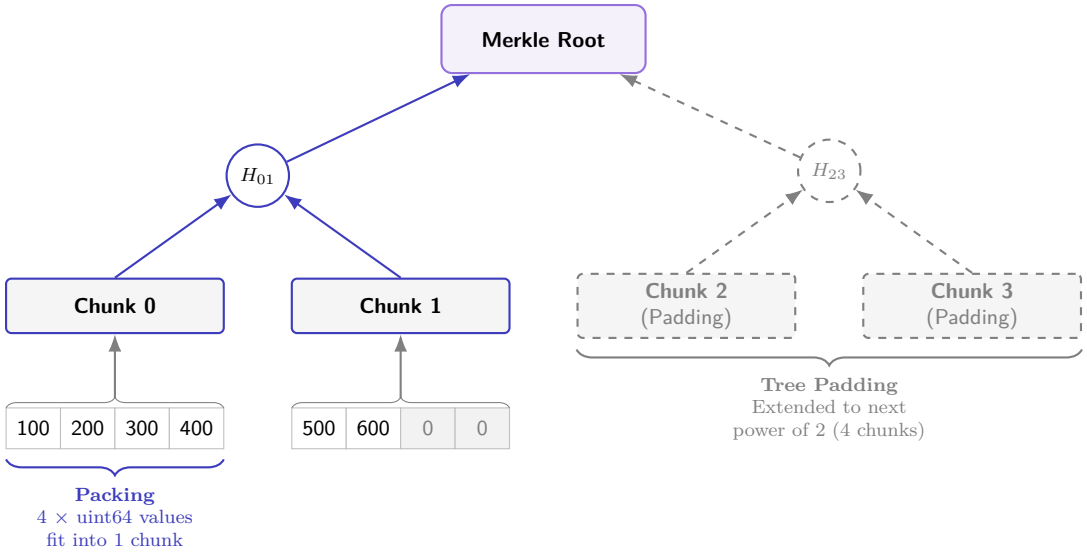

Figure 2.5: Packing and Merkleization. A list of six integers is packed into two chunks. The tree is then padded with zero-chunks to the next power of two (4 chunks) before hashing.

2.4.3 Mixing in the Length

For types with variable lengths, such as `List`, hashing the raw data alone is insufficient. A list containing values $[A, B]$ and a list containing $[A, B, 0]$ (where 0 is the default padding value) would produce identical Merkle trees if only the data chunks were hashed. This creates a collision vulnerability.

To resolve this, SSZ employs a mechanism called Mix-in Length. After the Merkle root of the actual data is computed, it is hashed together with the number of elements in the list. This ensures that lists with different lengths always result in different final roots, even if their data contents appear identical due to padding. This process is detailed in Figure 2.6.

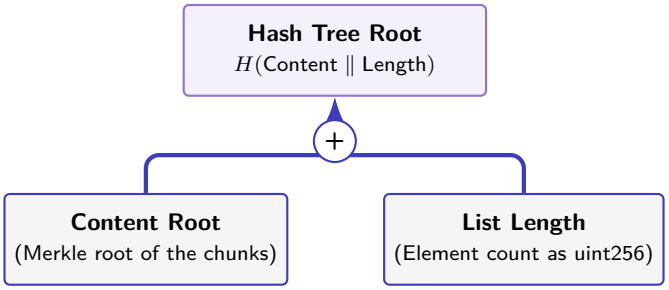

Figure 2.6: The Mix-in Length mechanism. The final identity of a List is a composite of its data and its size, preventing length-extension ambiguity.

2.5.0 Generalized Indices and Proofs

Generalized indices provide a deterministic addressing scheme for every piece of data within an SSZ structure. By mapping the data structure to a binary Merkle tree, we can assign a unique

integer index to every node, from the root down to the individual leaves.

The addressing follows a simple recursive rule:

- The root of the tree is at index 1.
- For any node at index k, its left child is at index $2k$.
- For any node at index k, its right child is at index $2k + 1$.

This numbering scheme has a useful property: the binary representation of an index describes the exact path from the root to that node. If we ignore the leading 1 (which represents the root), every subsequent bit serves as a navigation instruction: 0 means "go left" and 1 means "go right."

Figure 2.7 illustrates this navigation logic. To find the element at index 6 (binary 110_2), we start at the root (1), move right (1), and then move left (0).

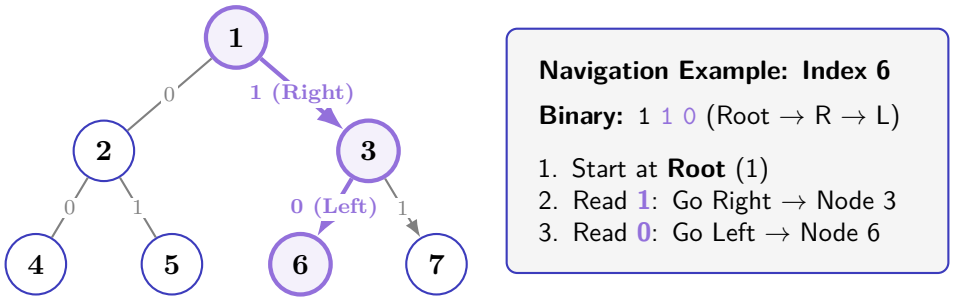

Figure 2.7: Generalized Indices. The index integer encodes the path from the root. Index 6 corresponds to the path Right-Left ($1 \to 0$) from the root.

This addressing system transforms the beacon chain state from a monolithic blob of data into a queryable database. A light client can verify a specific piece of data (such as a single validator's balance) without possessing the entire state. The full node provides the specific leaf at index k and a Merkle Multiproof—the minimal set of sibling hashes required to reconstruct the root. If the computed root matches the trusted network root, the light client can be mathematically certain that the data is authentic.

2.5.1 Example: Verifying a Validator Balance

To demonstrate how these indices enable efficient verification, consider a `Validator` container. This structure holds four specific fields:

1. `pubkey` (48 bytes)
2. `withdrawal_credentials` (32 bytes)
3. `effective_balance` (uint64)
4. `slashed` (boolean)

Since the container holds exactly four fields, these elements form the leaves of a binary tree with a depth of 2 (because $2^2 = 4$).

In the generalized index system, the indices for a specific layer d always start at 2^d. Therefore, our four fields are mapped to the continuous range of indices starting at 4 $(4, 5, 6, 7)$.

To determine the exact index of the `effective_balance`, we calculate the offset from the start of the layer. Since `effective_balance` is the third field, its zero-indexed position is 2. We combine these values as follows:

$$\text{Generalized Index} = \underbrace{2^{\text{depth}}}_{\substack{\text{Layer Start Index} \\ (2^2=4)}} + \underbrace{\text{Field Position}}_{\substack{\text{3rd field} \\ \text{(offset 2)}}} = 4 + 2 = 6$$

Figure 2.8 visualizes the resulting Merkle Multiproof. To verify the balance, a light client does not need the entire validator record. It only requires the target value (Index 6) and the specific *sibling hashes* along the path to the root (Index 7 and Index 2).

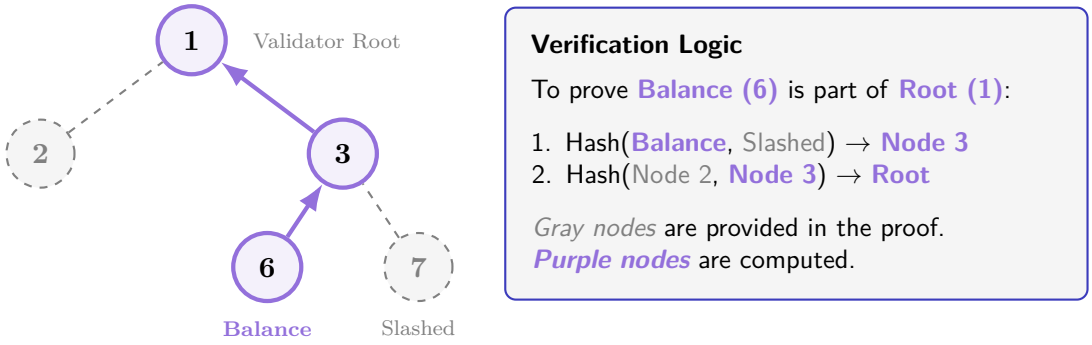

Figure 2.8: Merkle Proof for a Validator's Balance. To verify the field at Index 6, the proof must provide the sibling hashes at Index 7 and Index 2. The verifier hashes these up the tree to reconstruct the Root.

3.0 The time model

Distributed systems face a basic constraint: there is no global clock that all participants can read. Each node has its own local time, clocks drift, and messages arrive with unpredictable delays. As a result, questions like "which message came first?" do not have a universal answer at the network level.

Consensus protocols avoid relying on fine-grained timestamps. Instead, they introduce a discrete time model: the protocol defines a sequence of fixed-duration windows called *slots*. Protocol rules are expressed in terms of slot numbers, rather than in terms of real-time ordering. The only wall-clock quantity that nodes must share is a single anchor point, the *genesis time*, which marks the beginning of slot 0.

Lean Consensus uses a short slot duration and further subdivides each slot into smaller *intervals*. The purpose of this chapter is to make this time model precise: what slots and intervals are, how they are computed from wall-clock time, and why the protocol benefits from structuring time this way.

3.1.0 Slots

A *slot* is the protocol's fundamental unit of time. Slots are numbered $0, 1, 2, \ldots$ and advance deterministically. At any wall-clock time, every node can compute the current slot number locally from the genesis time and the slot duration. Figure 3.1 provides the basic picture: slots form a uniform grid laid over real time.

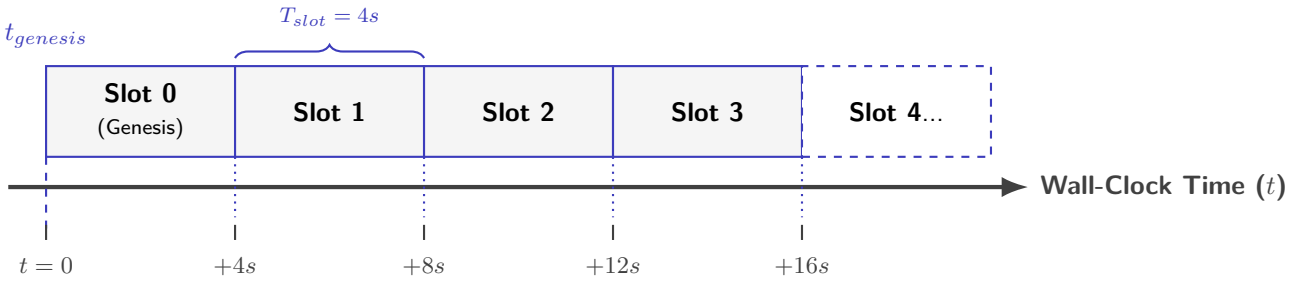

Figure 3.1: The relationship between physical wall-clock time and protocol slots. The genesis time $t_{genesis}$ anchors the system. Every 4 seconds, the protocol deterministically transitions to the next slot number.

This discretization does not remove network delays, but it changes how they are handled. Instead of trying to totally order all events by timestamps, the protocol mostly cares about which slot an event belongs to. Events within the same slot are treated as belonging to the same time window,

which absorbs small timing differences and avoids turning clock skew into consensus ambiguity.

3.2.0 Slot duration and intervals

Lean Consensus fixes two public parameters:

- Slot duration: $T_s = 4$ seconds.
- Interval duration: $T_i = 1$ second.

Each slot is subdivided into $T_s/T_i = 4$ intervals, numbered $0, 1, 2, 3$. The interval structure gives the protocol a shared schedule for when particular messages are expected to be produced and when they are allowed to influence fork choice. Figure 3.2 shows one slot as four consecutive one-second phases.

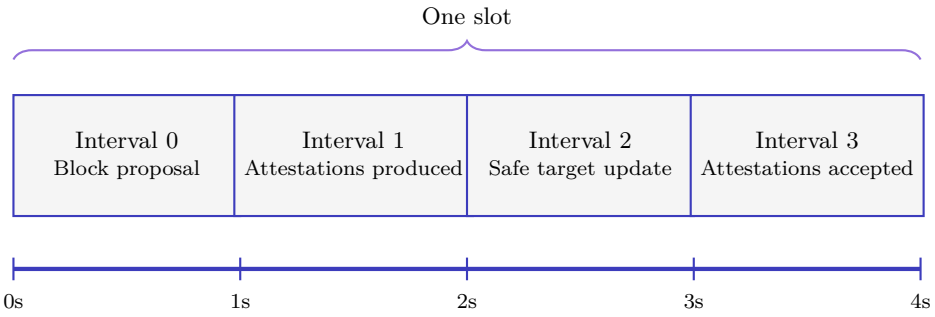

Figure 3.2: A slot is subdivided into four one-second intervals. Intervals define when different protocol actions occur.

The reason to stage actions like this is that time itself can become adversarial. In proof-of-stake protocols, validators can only attest to blocks they have actually received, and attestations must be produced before a fixed deadline within the slot. As a result, the moment at which a block is published determines how many validators see it in time to vote for it.

This dynamic is illustrated in Figure 3.3. A block proposer may benefit from delaying publication slightly, because waiting allows them to incorporate more information or extract more value. However, delaying also reduces the time available for the block to propagate through the network. The proposer is therefore trading off additional value against the risk that too few validators attest to the block. These strategic timing decisions, and their negative effects on consensus stability, are known in the Ethereum research literature as *timing games* [2].

A common mitigation pattern is therefore to make timing explicit and coarse-grained. Rather than letting messages influence consensus as soon as they arrive, the protocol defines clear phases within each slot and delays the point at which attestations begin affecting fork choice. This gives honest validators a predictable window to receive and verify blocks, and reduces the benefit of fine-grained message timing.

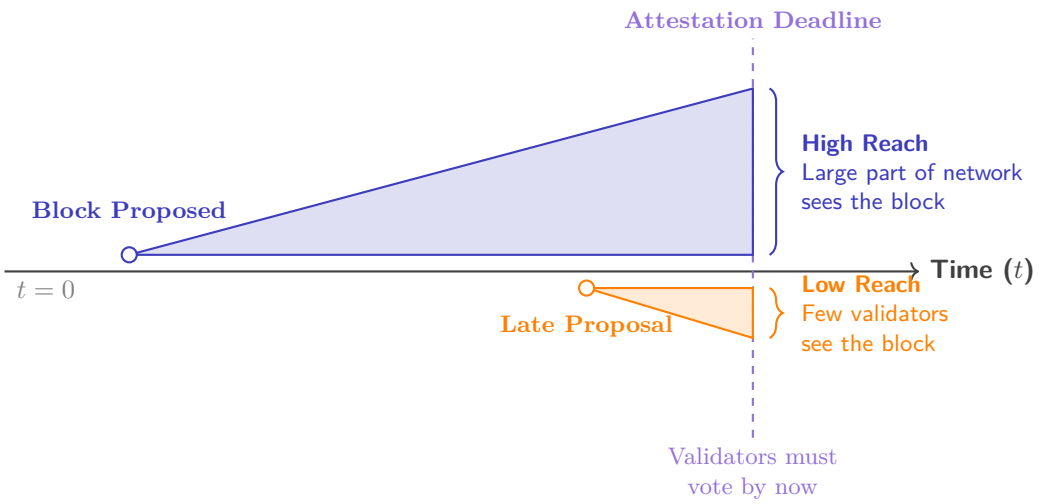

Figure 3.3: The race against the deadline. The vertical axis represents the percentage of the network that has received the block. An early proposal (blue) allows the gossip protocol to reach nearly all validators before the deadline. A late proposal (orange) leaves insufficient time for propagation, resulting in missed votes and potential consensus failure.

3.2.1 The four-interval workflow

The interval structure, as described in Figure 3.3, creates a pipeline that processes blocks and votes in a predictable sequence. This separation ensures that the network has time to reach a stable state before decisions are finalized.

1. **Interval 0: Block Proposal.** The slot begins with the publication of new data. The validator designated as the *proposer* creates a block and broadcasts it to the network.

   Crucially, the proposer also casts their own vote (attestation) immediately. Instead of sending two separate messages—one for the block and one for the vote—the proposer packages their vote directly inside the block envelope (the data structure containing the block header and body). This optimization ensures the block comes with at least one vote of support attached.

2. **Interval 1: Attestation Broadcast.** Once the block is propagated, the remaining validators (the *attesters*) inspect it. If the block is valid, they broadcast their attestations.

   At this stage, these incoming votes enter a pending state. The node stores them in memory but does not yet apply them to the consensus algorithm. Effectively, the node acknowledges receipt of the votes but does not yet allow them to influence the choice of the best chain.

3. **Interval 2: Safe Target Update.** Before counting the new votes, the protocol stabilizes its view of the chain history. It computes the safe target—the most recent block that has gathered a supermajority (at least 2/3) of the total stake.

   The safe target acts as a high-confidence anchor. By identifying which block is "safe" before processing new votes, the protocol ensures that subsequent voting is built upon a stable foundation, reducing the likelihood of chaotic reorganizations.

4. **Interval 3: Attestation Acceptance.** Finally, the pipeline completes. The attestations that were held in the pending state during Interval 1 are promoted to known status.

The protocol now feeds these votes into the fork choice rule. The weights of the blocks are updated, and the node recomputes the canonical head—the tip of the chain with the most accumulated weight. This new head becomes the parent for the next slot's proposal.

3.3.0 The slot clock

To make slot time concrete, the protocol needs a deterministic conversion from wall-clock time to a slot number and an interval number. Let:

- t_g be the genesis time (a Unix timestamp, in seconds),
- t be the current Unix time (in seconds),

The current slot is computed as:

$$\text{slot}(t) = \begin{cases} 0 & \text{if } t < t_g, \\ \left\lfloor \dfrac{t - t_g}{T_s} \right\rfloor & \text{otherwise.} \end{cases} \tag{3.1}$$

Within the current slot, define the offset into the slot:

$$\Delta(t) = (t - t_g) \bmod T_s, \tag{3.2}$$

and the corresponding interval:

$$\text{interval}(t) = \left\lfloor \frac{\Delta(t)}{T_i} \right\rfloor \in \{0, 1, 2, 3\}. \tag{3.3}$$

Figure 3.4 summarizes the flow from wall-clock time to protocol time.

3.3.1 Slot zero: the genesis slot

Slot zero holds a special place in the protocol. It is the *genesis slot*—the origin point of the chain. Unlike all subsequent slots, slot zero has no proposer and no parent block. The genesis state is created directly from configuration rather than derived from a previous state.

The genesis block serves as the trust anchor for the entire system. It is automatically considered both justified and finalized, bootstrapping the consensus mechanism. When the first real block arrives (at slot 1 or later), special logic acknowledges the genesis block as the immutable starting point from which all chain history flows.

3.4.0 Shorter slot rationale: The push for reduced latency

The decision to target a 4-second slot duration represents a significant acceleration from Ethereum's historical 12-second heartbeat. While recent proposals such as EIP-7782 have explored conservative reductions to 6 or 8 seconds to improve throughput and user experience [3], Lean Consensus sets an aggressive engineering target of 4 seconds. This shift is motivated by three primary economic and structural factors identified in recent market microstructure research.

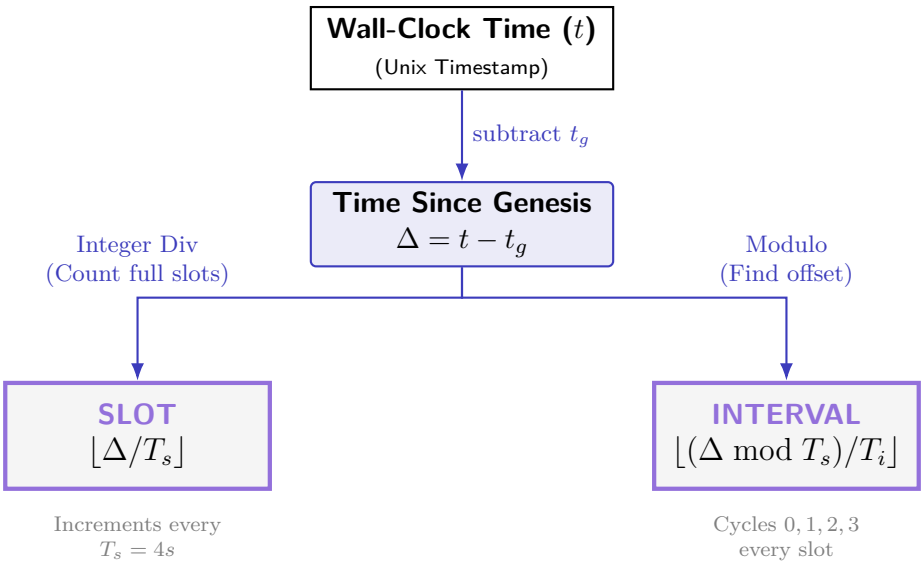

Figure 3.4: The slot clock logic. Wall-clock time is first normalized relative to genesis, then split into two coordinates: the Slot Number (coarse time) and the Interval (fine-grained offset within the slot).

1. **The Settlement Layer (Confirmation Latency).** Ethereum's primary role has evolved into a settlement layer for Layer 2 (L2) rollups. Rollups rely on the L1 chain to finalize their state or, at minimum, provide a high-confidence safe head—a block that has received a supermajority of validator votes and is thus statistically unlikely to be reorganized, even before full finality. The current 12-second latency forces cross-chain bridges and arbitrageurs to wait significant periods to ensure a transaction cannot be reorged. Reducing the slot time linearly reduces the time-to-inclusion and accelerates the generation of this safe confirmation signal that L2s read to derive the canonical state [4].

2. **Economic efficiency (reducing price staleness).** On-chain automated market makers (AMMs) update their prices only when a new block is produced. Between blocks, prices remain fixed, even if the true market price of the asset moves elsewhere. By contrast, centralized exchanges update prices continuously as orders arrive.

   This mismatch creates a predictable pattern of value extraction. When the market price moves between two blocks, arbitrageurs can trade against the AMM at the beginning of the next block, buying or selling at a price that is known to be outdated. These trades are risk-free for the arbitrageur and the resulting losses are borne by the AMM's liquidity providers.

   This effect is known as Loss-Versus-Rebalancing (LVR) [5]. Importantly, the magnitude of LVR depends on how long prices remain stale. Shorter slot times reduce the maximum price drift that can occur between blocks, shrinking the arbitrage opportunity available at each block boundary. Recent work shows that reducing this effect lowers the effective cost of providing liquidity on-chain, which can in turn support deeper and more efficient markets [6].

3. **Resource Smoothing.** For a fixed transaction throughput (gas per second), shorter slots imply smaller block payloads. Propagating smaller payloads more frequently smooths out bandwidth spikes compared to propagating massive payloads infrequently. This reduction in peak bandwidth load helps maintain P2P network health and accessibility for nodes with constrained connections, even as total data throughput requirements grow [3].

3.5.0 Rationale: Explicit intervals and timing games

While shorter slots offer significant economic and utility benefits, they impose stricter requirements on the network. Specifically, reducing the slot time reduces the safety margin available for block propagation, making the consensus mechanism highly sensitive to *timing games*.

In a naive protocol, a block proposer is incentivized to delay their block publication as late as possible into the slot. By waiting, the proposer accumulates more order flow (MEV) and can capture more value. However, if a proposer broadcasts their block too late, honest validators may not receive it in time to verify and attest to it before the deadline. This behavior destabilizes the consensus layer, leading to missed slots and potential reorgs [2].

The rigid interval schedule described in Figure 3.2 acts as a counter-measure. By strictly enforcing a proposal deadline (Interval 0) separate from the attestation deadline (Interval 1), the protocol imposes a hard temporal boundary on block validity. If a proposer delays publication beyond the designated sub-slot to accumulate additional MEV, honest validators reject the block as untimely, thereby neutralizing the incentive for strategic delays and safeguarding the integrity of the 4-second cycle.

4.0 The State Transition Function

At the core of the consensus protocol lies a single, deterministic mathematical function. Regardless of the complexity of the underlying networking or cryptography, the fundamental operation of the blockchain can be summarized by the state transition function, denoted as Υ:

$$S_{n+1} = \Upsilon(S_n, B)$$

where S_n is the system state at slot n, and B is the incoming block.

This chapter examines the mechanical process of verifying a block and applying it to the current state. We do not strictly address how the network achieves agreement on which block is canonical (consensus); rather, we focus on the validity conditions that any proposed block must satisfy to be accepted by an honest node.

4.1.0 The System State

The state is the single source of truth for the consensus protocol. It serves as a comprehensive snapshot of the system at a discrete moment in time.

To understand why the state exists, consider the validation problem: when a node receives a new block, it must determine if that block is valid. To do this without looking back at the entire history of the chain, the node needs a summary of everything that has happened up to that point. The state is that summary.

It contains exactly the information required to validate the next block, and nothing more. As summarized in Figure 4.1, the state is divided into three critical domains: identity (who can act), chronology (what has happened), and active voting (what is happening now).

4.1.1 The Validator Registry

The first major component of the state is the validator registry. Think of this as the master membership list for the network. In a permissionless system where anyone can join, the protocol needs a strict way to track exactly who is currently allowed to participate. The registry handles three specific problems:

- **Authentication:** The registry stores the public key for every validator. When the network receives a block or a vote signed by a validator, it uses this key to cryptographically verify that the message is authentic and hasn't been forged.

- **Authorization:** Just because a validator exists doesn't mean they are currently active. The registry tracks a status for each participant (e.g., *Active*, *Exited*, or *Slashed*). The protocol uses this to ignore messages from validators who have withdrawn or been penalized.
- **Weighting:** In Proof-of-Stake, one validator does not necessarily equal one vote. Influence is proportional to the amount of ETH staked. The registry tracks the effective balance of each validator. This number determines the weight of their vote.

4.1.2 The Chain Context

The second major component of the state is the chain context. If the registry tracks the *participants*, the context tracks the *timeline*. To validate a new block, the system needs to know exactly where it fits in history. The state maintains this continuity through three mechanisms:

- **The Clock:** The state stores the current slot number. This acts as the heartbeat of the system. It ensures strict ordering: if the state is currently at Slot 100, it knows that a block for Slot 101 is the logical next step, while a block for Slot 90 is outdated and irrelevant.
- **The History Buffer:** Every block must connect to a parent block to form a chain. To verify this connection quickly, the state keeps a short-term memory of recent block hashes (roots). When a new block claims "My parent is Block X," the protocol checks this buffer to confirm that Block X actually exists and was recently accepted.
- **The Safety Anchor:** In a distributed network, the most recent few blocks can sometimes change (due to temporary disagreements). However, older history must remain permanent. The state tracks the latest finalized checkpoint. This acts as a solid floor: the chain can grow upward from this point, but the protocol will verify that no new block attempts to rewrite history below this line.

4.1.3 Active Voting

Finally, the state serves as the active scoreboard for the current consensus round. While the other components track established history, this section handles the immediate present.

It actively aggregates incoming votes against candidate blocks to determine if they have enough support. To efficiently track this progress, the state maintains two synchronized components:

- **The Candidate List:** A record of the specific block roots that are currently receiving votes. These are the potential next steps vying to become a permanent part of the chain's history.
- **The Voter Manifest:** A compact bitfield tracking exactly which validators have cast a vote for each candidate. By summing the effective balances of these validators, the protocol determines if a candidate has reached the supermajority threshold.

4.2.0 Block Anatomy

The block is the vehicle for changing the blockchain's state. To make verification efficient for different types of users (like light clients versus full nodes), the block structure is strictly divided into two parts: a lightweight header and a heavy body, as illustrated in Figure 4.2.

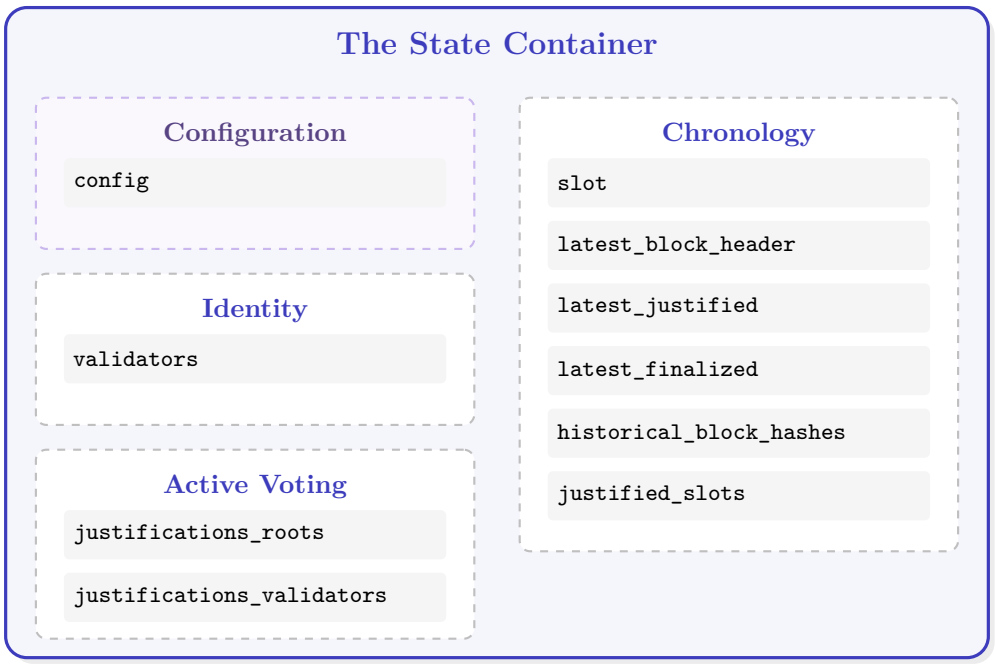

Figure 4.1: The Architecture of the State. The state object aggregates four distinct types of data: static configuration, the validator registry, historical context, and the active consensus scratchpad.

4.2.1 The Block Header

The header is a small, fixed-size summary of the block. Think of it as the envelope that contains the metadata about the message inside. Because it is so small, any node can download and verify the headers of the entire chain very quickly without needing to download the actual transactions. The header contains critical fields that anchor the block in history:

- **Slot:** This integer indicates exactly *when* this block belongs in the timeline. Since the protocol advances in fixed heartbeats, the slot number tells us the block's intended position in the schedule.
- **Proposer Index:** The ID of the validator assigned to create this block. This allows honest nodes to verify that the block was produced by the correct participant for this specific time slot.
- **Parent Root:** This is the cryptographic hash of the previous block's header. By including this, the new block explicitly points to its ancestor, extending the chain. If you change a single bit in a past block, its hash changes, breaking this link in all future blocks.
- **State Root:** This is the fingerprint of the entire system state *after* this block has been executed. It allows the proposer to claim: "If you run the transactions in this block against the current state, the result will look exactly like this." This is a way for other nodes to verify the computation.
- **Body Root:** This is a hash of the block body (the actual payload). It acts as a unique identifier for the set of transactions and votes included in the block. It ensures that the content of the block cannot be altered without invalidating the header.

4.2.2 The Block Body

The block body is the container for the variable-length protocol payload. While the header is optimized for constant-size verification (light clients), the body contains the full data required for state execution.

The body consists primarily of a list of attestations. These are the aggregated votes from validators for a specific slot. Since the number of attestations varies based on network participation and aggregation efficiency, the size of the body is dynamic.

To bind the payload to the metadata, the protocol enforces a strict cryptographic commitment: the entire body is serialized and hashed. This hash is stored in the header's body root field. This mechanism ensures that the header acts as a unique, tamper-evident identifier for the specific payload it accompanies.

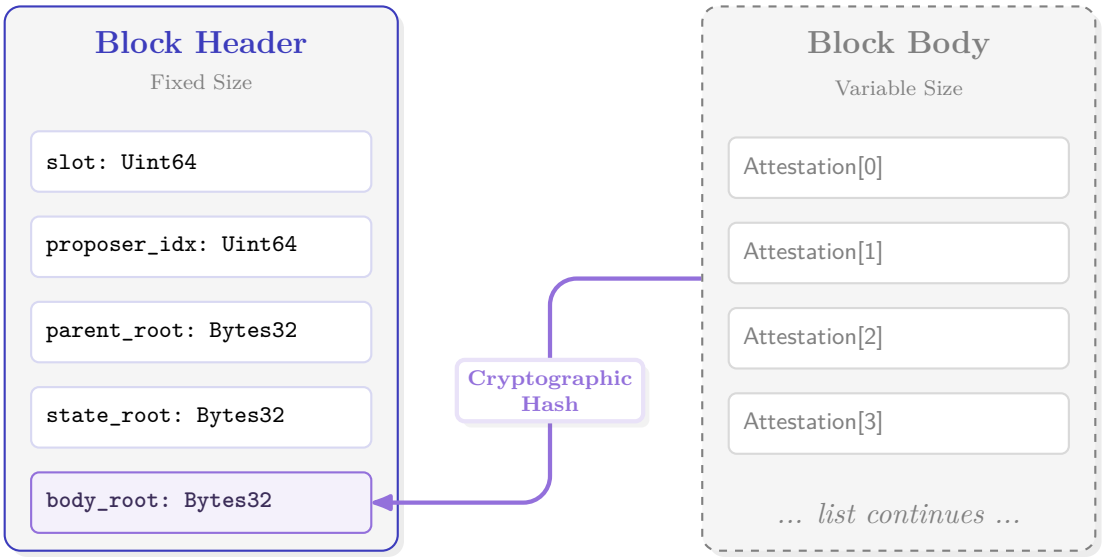

Figure 4.2: The cryptographic commitment. The header contains metadata and the body root, which is a hash of the variable-length body. This allows the header to serve as a constant-size proxy for the full block data.

4.3.0 The Transition Pipeline

The state transition is a strict, sequential pipeline. To prevent the network from diverging, every node must apply the exact same set of operations to every block. If a block fails any stage of this pipeline, it is effectively invisible to the protocol: the state remains unchanged, and the block is discarded.

We can divide this process into four distinct phases: time synchronization, structural validation, consensus execution, and integrity verification.

4.3.1 Phase 1: Slot Processing

The protocol enforces a continuous flow of time, but blocks arrive sporadically. A block proposed for slot N may arrive when the state is still at slot $N - 5$. Before the block can be processed, the state must be advanced to the same point in time.

This gap is filled by processing empty slots (or skipped slots). As illustrated in Figure 4.3, the system iteratively advances the state one slot at a time until it matches the incoming block. For each empty slot, the system performs two critical maintenance tasks:

1. **History Freezing:** The current state root is calculated and frozen into the historical accumulator. This ensures that future proofs can reference the state as it existed at this specific moment, even if no block was produced.

2. **Clock Advance:** The state's slot counter is incremented.

This ensures that the state is perfectly synchronized with the block's timeline before any execution logic begins.

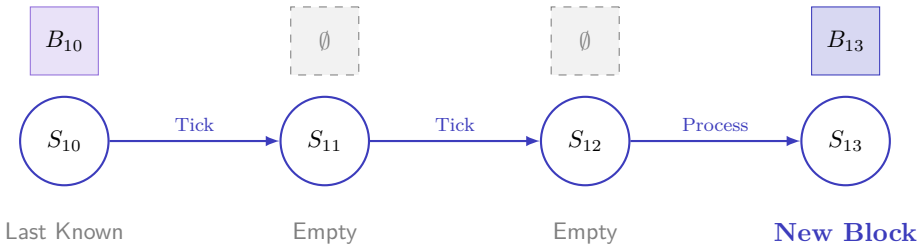

Figure 4.3: Processing empty slots. The state must conceptually tick through empty slots 11 and 12 before block 13 can be applied.

4.3.2 Phase 2: Header Validation

Before verifying the heavy payload, the system validates the fixed-size header. This acts as a computationally cheap filter to reject invalid blocks early. The checks focus on authorization and continuity, as summarized in Figure 4.4:

- **Proposer Identity:** The system calculates the expected proposer for the current slot using the registry. It verifies that the proposer index in the header matches this expectation.

- **Parent Linkage:** The block's parent root must match the hash of the latest block header recorded in the state.

- **Chronology:** The block's slot must be strictly greater than its parent's slot.

If these checks pass, the state updates its historical accumulators (appending the parent root and extending the justification bitfield) to prepare for the new block.

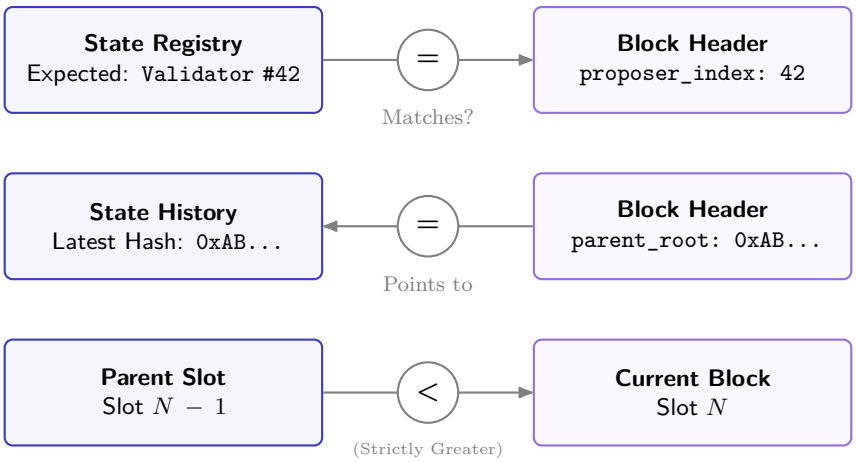

Figure 4.4: The three structural checks of the header validation phase. The incoming block (purple) is compared against the local state (blue) to ensure authorization, continuity, and chronological order.

4.3.3 Phase 3: Payload Execution (Consensus)

The block body carries the network's votes, known as attestations. This phase executes the core consensus logic. As shown in Figure 4.5, the system iterates through every attestation in the list and performs four logical steps:

1. **Vote Validation:** First, the system checks if the vote is legal. The vote must originate from a checkpoint that is already *justified* (trusted). Additionally, the vote must target a slot that is eligible to become a new checkpoint. If these conditions are not met, the vote is ignored.

2. **Weighting:** If the vote is valid, the protocol counts it. In Proof-of-Stake, votes are weighted by value, not by count. The system adds the validator's *effective balance* to the total weight for the target block.

3. **Justification:** The system checks if the target block has gathered enough support. If the total weighted votes exceed 2/3 of the total active stake, the block is marked as justified. It is now considered safe and can serve as a source for future votes.

4. **Finalization:** Finally, the system checks for continuity. If there is an unbroken chain of justified checkpoints connecting the old finalized point to the newly justified point, the system advances the finalized checkpoint to this new location. All history prior to this boundary becomes permanently immutable.

4.3.4 Phase 4: State Root Verification

After the body is processed, the state object has been mutated into a new version, S_{n+1}. The final step is to verify integrity. The protocol computes the hash tree root of this new state object:

$$R_{\text{computed}} = \text{HashTreeRoot}(S_{n+1})$$

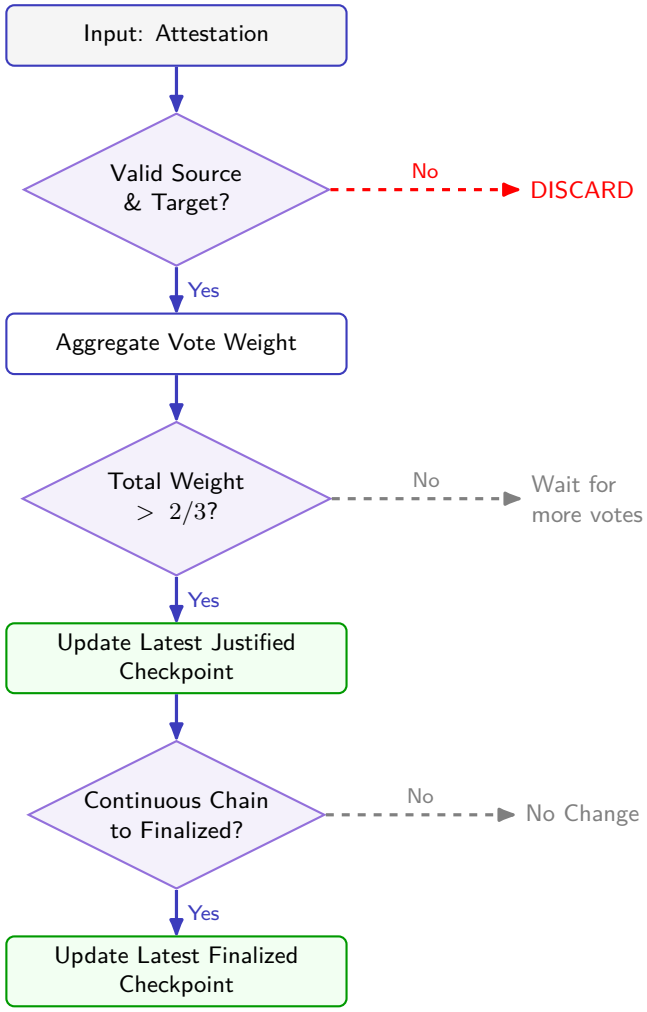

Figure 4.5: The consensus logic flow. For every attestation in the block, the system validates it, tallies the weight, and checks if it triggers a justification or finalization update.

This computed root is compared against the state root declared in the block header:

$$R_{\text{computed}} \stackrel{?}{=} B.\text{state\_root}$$

If they match, the block is accepted. If they differ, it implies the proposer executed the transition differently than the verifier (or is lying about the result). In this case, the block is invalid and rejected. The entire validation pipeline is summarized in Figure 4.6.

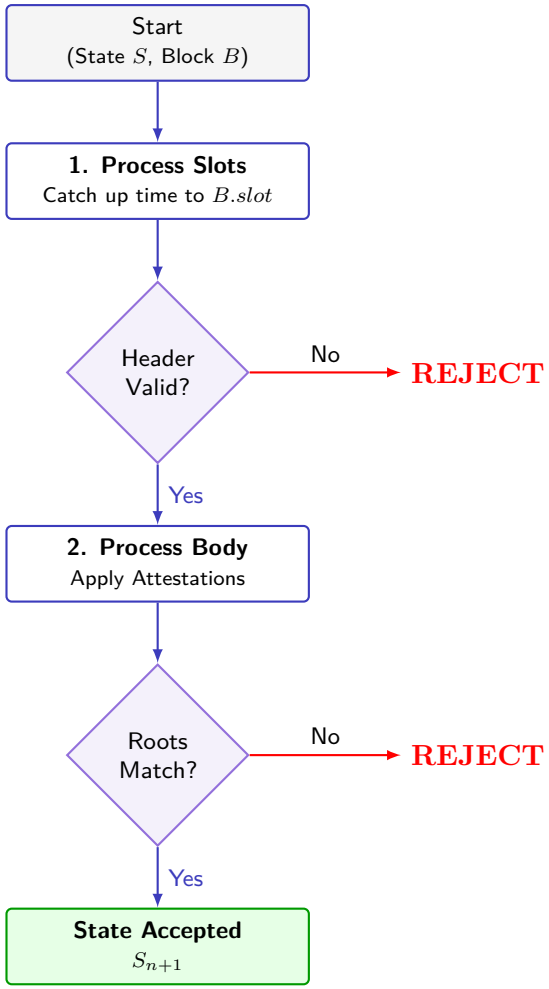

Figure 4.6: The block verification pipeline. Validation ensures that only blocks satisfying all protocol rules are applied to the local state.

5.0 The Peer-to-Peer Layer

The consensus mechanisms described in previous chapters focus on how nodes agree on the truth. However, consensus cannot function in isolation; it requires a constant flow of data. When a validator produces a block, that block must physically travel to thousands of other computers around the world.

In a centralized system, this would be handled by a master server that relays messages between clients. Ethereum, by definition, eliminates this central point of control. Instead, it relies on a Peer-to-Peer (P2P) layer where every node acts as both a client and a server, contributing its own bandwidth to propagate information across the mesh.

This layer handles the logistics of networking: finding other computers, establishing secure connections, and routing data. To understand how it operates, it is helpful to distinguish between three specific problems the protocol must solve:

1. **Discovery:** When a node first comes online, it is isolated. It needs a mechanism to find other participants on the internet who speak the Ethereum protocol and are synchronizing the same chain.
2. **Transport:** Once a peer is found, the node must establish a connection. The transport layer handles the handshake, encryption, and authentication, ensuring that the two machines can exchange bytes securely without interference.
3. **Application Protocols:** Finally, the nodes must speak a common language. This involves distinct sub-protocols for different tasks, such as gossiping new blocks to the entire network or requesting specific historical data to sync the chain.

A major challenge in building this system is that the network is dynamic. Nodes are not permanent fixtures; they join and leave the network at will, and their physical network addresses (IP addresses) frequently change. If the system relied solely on IP addresses to identify peers, the network would quickly become disconnected as contact information became stale.

To address this, Ethereum uses a specialized data structure to decouple a node's persistent identity from its temporary location. We begin this chapter by examining this structure, known as the Ethereum Node Record.

5.1.0 Ethereum Node Records (ENR)

In early distributed systems, a node's identity was often conflated with its network location (an IP address and port). This approach is brittle; if a node moves to a different network or restarts with a new IP, it effectively becomes a stranger to its peers, losing established trust and connectivity.

Ethereum solves this by strictly decoupling identity from location using the Ethereum Node Record (ENR), defined in EIP-778 [7]. An ENR acts as a self-certifying document—a cryptographic business card—that binds a stable identity (a public key) to dynamic information (IP addresses, ports, and capabilities).

5.1.1 The Envelope Structure

Internally, an ENR is an RLP (Recursive Length Prefix) list containing a sequence of data fields. While the format allows for arbitrary key-value pairs, the protocol strictly enforces the structure of the envelope to ensure security, freshness, and compatibility. The record consists of three distinct parts, strictly ordered:

1. **Signature:** A 64-byte cryptographic signature over the record's content. This signature ensures the integrity and authenticity of the data, preventing tampering by malicious actors.
2. **Sequence Number:** A 64-bit unsigned integer that tracks versions. Whenever a node updates its information (e.g., its IP changes or it supports a new protocol), it increments this number and re-signs the record. Peers use this number to determine which record is the most current.
3. **Identity Scheme:** A mandatory key-value pair (key `id`) that defines the cryptographic schema used. For Ethereum, this is typically `v4`, which implies the presence of a `secp256k1` public key in the content. This scheme tells peers how to verify the signature and derive the node's unique Node ID.
4. **Content:** A list of arbitrary key-value pairs, sorted lexicographically by key. Beyond the identity keys, this includes network endpoints (IP addresses, ports) and protocol-specific metadata (like the `attnets` bitfield).

The entire encoded record is capped at a maximum size of 300 bytes. This constraint is intentional: it ensures that an ENR fits comfortably inside a single UDP packet, which is critical for the efficiency and reliability of the low-level discovery protocol (Discv5).

Figure 5.1 illustrates the logical layout of the record, highlighting the separation between the signature and the signed content.

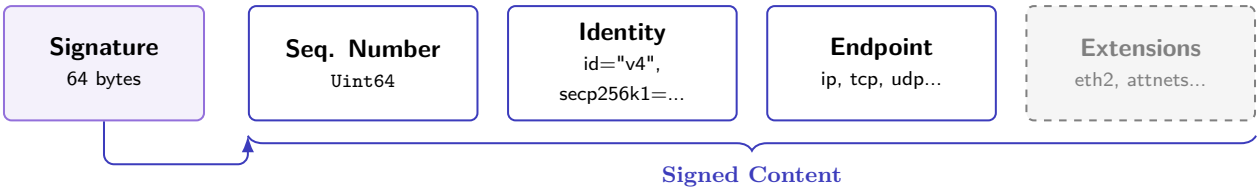

Figure 5.1: The ENR structure. The signature covers the sequence number and all key-value pairs. The key-value pairs provide flexibility, allowing the record to store generic network info (IP/UDP) alongside consensus-specific data.

5.1.2 Identity Schemes

One of the design goals of the ENR format is cryptographic agility. Hardcoding a specific signature algorithm (like ECDSA) or hash function (like SHA-256) would make the network difficult

to upgrade in the future—for example, if quantum computers render current elliptic curves insecure.

To solve this, the protocol abstracts these choices into an Identity Scheme. Every record must contain a key named `id` whose value specifies which set of cryptographic rules to apply. This value acts as a dispatcher: it tells the receiving node how to verify the signature and how to derive the node's unique identifier.

Currently, the Ethereum network uses a single scheme named "v4". It is defined as follows:

- **Cryptography (The Keys):** The "v4" scheme relies on the same elliptic curve cryptography used for Ethereum accounts and transactions: secp256k1.

  Inside the record, the node's public key is stored in a compressed format (33 bytes) under the key `secp256k1`. Compression is used here to save space—a critical resource in UDP-based discovery protocols where every byte counts.

- **Node ID:** Every node needs a unique, permanent identifier so that peers can recognize it across sessions. In this scheme, the Node ID is not a random number assigned by a server; it is derived mathematically from the node's identity.

  Specifically, the Node ID is the Keccak256 hash of the *uncompressed* public key. This cryptographic binding ensures that a node cannot change its ID without changing its underlying keypair.

  $$\text{NodeID} = \text{Keccak256}(\text{PubKey}_{\text{uncompressed}})$$

- **Signing Rule:** To prove that the record is authentic, the node must sign it. The process, illustrated in Figure 5.2, ensures that the signature covers every meaningful part of the record except the signature itself.

  1. **Assembly:** The node gathers the sequence number and all key-value pairs into a list: `[seq, k1, v1, k2, v2, ...]`. Crucially, the keys are sorted alphabetically to ensure a deterministic order.
  2. **Serialization:** This list is encoded into bytes using RLP (Recursive Length Prefix), Ethereum's standard serialization format.
  3. **Signing:** The RLP bytes are hashed using Keccak256 to create a small, fixed-size fingerprint (digest). This digest is then signed using the node's private key to produce the final signature.

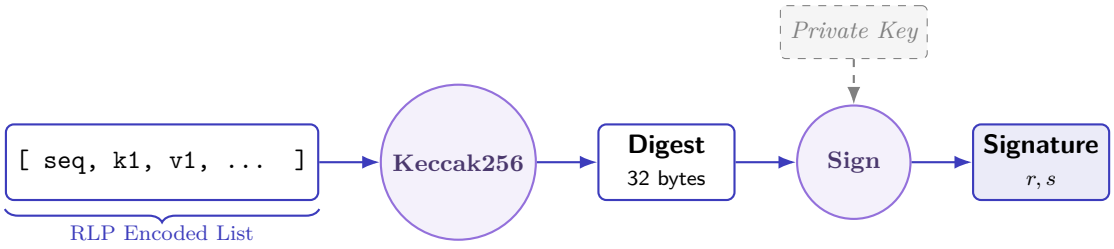

Figure 5.2: The ENR signing pipeline. The content list is RLP-encoded and hashed to create a fingerprint, which is then signed using the node's private key.

By relying on this `id` string, the network can introduce new schemes (e.g., "v5") in the future without breaking backward compatibility for existing nodes.

5.1.3 Consensus Extensions: Beyond IP Addresses

While EIP-778 defines the generic format for node records, the Ethereum consensus layer extends this format to solve a critical networking challenge: topology filtering.

In a global network with thousands of nodes, no single computer can connect to everyone. A node has a limited number of slots for peers (typically 50-100). If a node fills these slots with random peers—some on testnets, some running outdated software, others listening to irrelevant data channels—it becomes isolated from the information it actually needs.

To prevent this, consensus clients add specific keys to their ENR. These keys act as public advertisements of the node's interests and status, allowing peers to filter connections before opening them.

The `eth2` Key: Fork Compatibility

The most fundamental check in any P2P network is: "Are we on the same chain?". A node running Mainnet must never connect to a node running the Sepolia testnet. Similarly, if the network undergoes a hard fork (an upgrade), updated nodes need a way to distinguish themselves from those still running the old rules.

The `eth2` key solves this by encapsulating a 16-byte value that acts as a compatibility badge that is illustrated on Figure 5.3. It contains three specific fields:

- **Fork Digest (4 bytes):** This is the network's primary firewall. It acts as a unique fingerprint for the current chain, ensuring that a Mainnet node never accidentally connects to a testnet or an outdated fork. It is computed by mixing two values:

  1. The Genesis Validators Root: A static hash of the initial validator set, which uniquely identifies the chain's origin (e.g., distinguishing Mainnet from Sepolia).
  2. The Current Fork Version: A version number that increments with every hard fork.

  If two nodes have different Fork Digests, they are effectively speaking different languages and will immediately disconnect.

- **Next Fork Version (4 bytes):** This field allows nodes to look into the future. It contains the version number of the *upcoming* upgrade. By advertising this, a node signals that it has already downloaded the new software and is ready for the transition.

- **Next Fork Epoch (8 bytes):** It specifies the exact time (in epochs) when the future upgrade will activate. This allows updated nodes to find each other before the fork happens, ensuring a smooth handover when the network rules change.

This structure allows the network to partition itself cleanly during upgrades. When a hard fork occurs, updated nodes change their Fork Digest, naturally causing them to disconnect from outdated peers and form a new, upgraded mesh.

The Subnet Bitfields

Once a node finds peers on the correct chain, it faces a second problem: bandwidth efficiency. It is impossible for every node to listen to every message on the network. To solve this, Ethereum

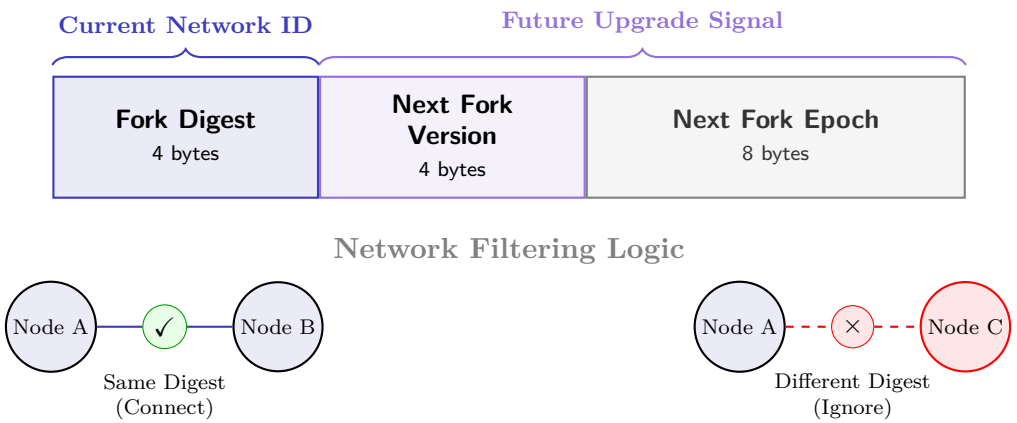

Figure 5.3: The `eth2` key structure. The first 4 bytes (Fork Digest) act as a strict firewall: if this value does not match, nodes simply ignore each other. The remaining 12 bytes allow nodes to coordinate future upgrades.

decouples the validator's duty from the network topology as illustrated on Figure 5.4.

1. **The Work (Committees):** Validators are mathematically divided into small working groups called *committees*. A validator assigned to a committee only needs to aggregate votes (attestations) from other members of that same group. It has no interest in the traffic of other committees.
2. **The Transport (Subnets):** To facilitate this, the P2P network is sliced into 64 separate data channels called subnets (subscription networks). Think of a subnet as a dedicated radio frequency.

The protocol defines a deterministic mapping between them: for any given slot, a committee is assigned to a specific subnet. This assignment rotates over time to distribute load.

Nodes use bitfields in their ENR to publicly advertise which of these channels they are currently listening to. A bitfield is simply a sequence of flags: if the N-th bit is set to 1, the node is subscribed to Subnet N, see Figure 5.4.

- **`attnets` (64 bits):** This field tracks the 64 subnets used for general consensus. If a node needs to find peers for a specific committee, it calculates the required subnet ID and searches the Discovery network for ENRs where that specific bit in `attnets` is set.
- **`syncnets` (4 bits):** This smaller field tracks the 4 specialized subnets used by the Sync Committee—a distinct, stable group of validators that allows light clients to verify the chain head with minimal overhead.

5.1.4 Updates and Freshness

Because an ENR is cryptographically signed, it is strictly immutable. Changing even a single bit of the IP address or port would invalidate the signature. Therefore, when a node changes its

configuration—for example, roaming to a new network or updating its listening port—it does not edit the existing record; instead, it generates an entirely new one.

To manage these replacements, the protocol relies on the Sequence Number (`seq`). This integer serves as the definitive version counter, allowing peers to immediately distinguish between current and outdated records.

The mechanism is straightforward: a record with a higher sequence number always supersedes one with a lower number. When a node updates its identity, it increments the sequence number by one, re-signs the content, and publishes the new ENR. Since there is no central authority to timestamp records, peers rely entirely on this value. If a node receives two conflicting records for the same Node ID, it implicitly discards the one with the lower sequence number and propagates the higher one. This ensures that the network eventually converges on the most up-to-date view of the node's identity without requiring global synchronization.

5.1.5 Text Encoding

While the ENR is transmitted as binary data (RLP) over the wire, engineers often need to share these records in configuration files, logs, or chat applications. To facilitate this, the standard defines a text-based representation.

The binary RLP bytes are encoded using URL-safe Base64 (to ensure the string is safe to use in web links and filenames) and prefixed with the tag `enr:`.

Example ENR String

`enr:-IS4QHCYrYZbAKWCBRlAy5zzaDZXJBGkcnh4MHcBFZntXNFrdvJjX04jRzjzCBOonrkTfj499SZuOh8R33Ls8RRcy5wBgmlkgnY0gmlwhH8AAAGJc2VjcDI1NmsxoQPKY0yuDUmstAHYpMa2_oxVtw0RW_QAdpzBQA8yWM0xOIN1ZHCCdl8`

Explanation: This string represents a node running on localhost. It decodes to:

- **IP:** 127.0.0.1
- **Port:** 30303 (UDP)
- **Key:** The associated secp256k1 public key.

The gibberish characters immediately following the `enr:` prefix contain the signature and the sequence number.

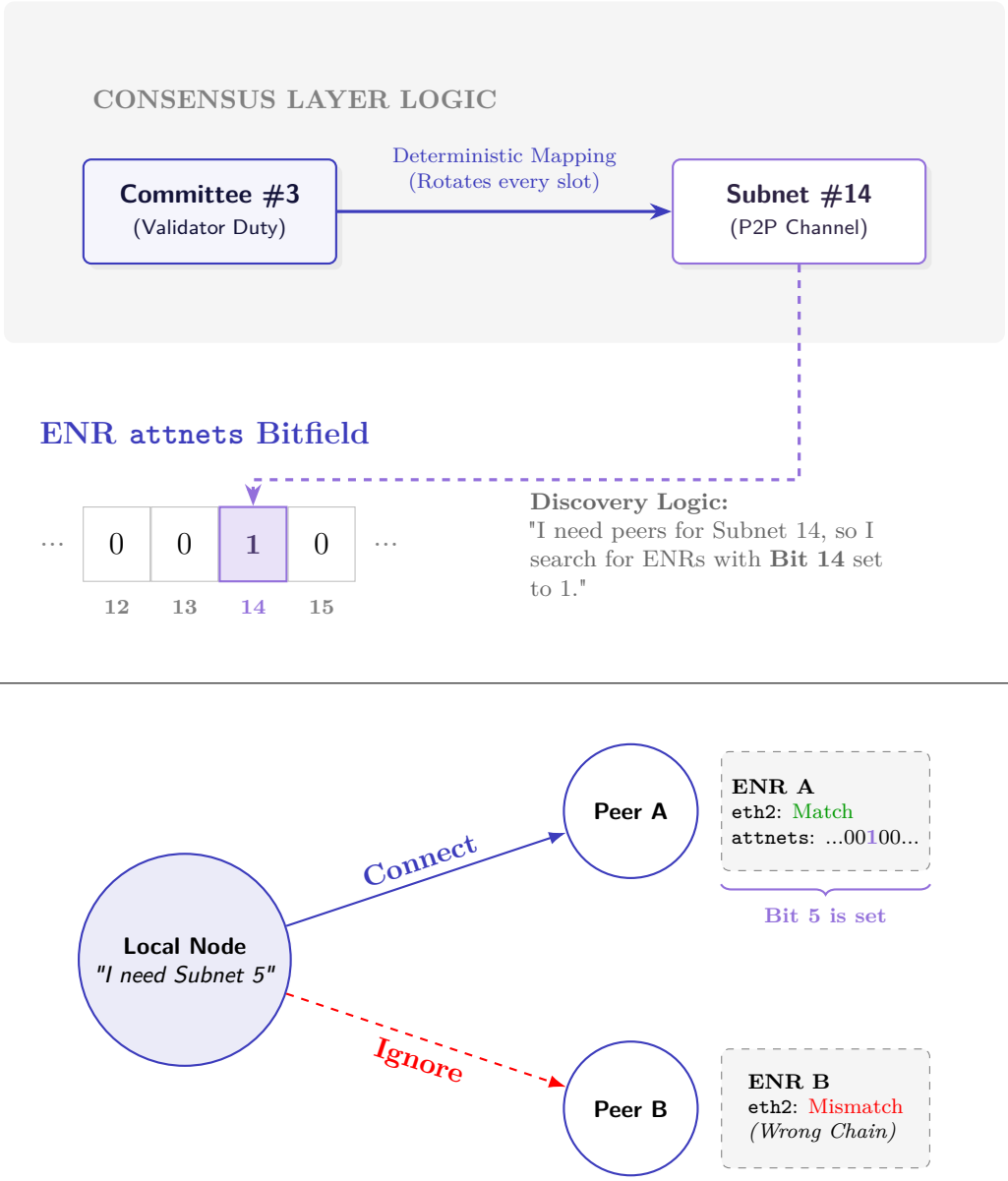

Figure 5.4: The complete Discovery lifecycle. **(Top)** How a validator determines which subnet bit to set in its ENR based on its committee duty. **(Bottom)** How a searching node uses those ENR bits (and the `eth2` field) to filter peers, connecting only to those on the correct chain and subnet.

Bibliography

[1] V. Buterin *et al.*, "Ethereum white paper," *GitHub repository*, vol. 1, no. 22-23, pp. 5–7, 2013.

[2] casparschwa and mike, "Timing games: Implications and possible mitigations," Ethereum Research (ethresear.ch), Dec. 2023, accessed: 2026-01-19. [Online]. Available: https://ethresear.ch/t/timing-games-implications-and-possible-mitigations/17612

[3] B. Monnot, B. Adams, D. Feist, and J. Ma, "Eip-7782: The case for 2x shorter slot times in glamsterdam," Ethereum Magicians Forum, Jun. 2025, accessed via provided context. [Online]. Available: https://ethereum-magicians.org/t/eip-7782-the-case-for-2x-shorter-slot-times-in-glamsterdam/24616

[4] V. Buterin, "Epochs and slots all the way down: ways to give ethereum users faster transaction confirmation times," Vitalik.ca, Jun. 2024. [Online]. Available: https://vitalik.eth.limo/general/2024/06/30/epochslot.html

[5] J. Milionis, C. C. Moallemi, T. Roughgarden, and A. L. Zhang, "Automated market making and loss-versus-rebalancing," *arXiv preprint arXiv:2208.06046*, 2022.

[6] J. Ma and D. Crapis, "The cost of permissionless liquidity provision in automated market makers," in *The International Conference on Mathematical Research for Blockchain Economy*. Springer, 2024, pp. 55–69.

[7] F. Lange, "EIP-778: Ethereum node records (enr)," Ethereum Improvement Proposals, Nov. 2017, draft. [Online]. Available: https://eips.ethereum.org/EIPS/eip-778