Lean Consensus

Contents

Chapter 1

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 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.

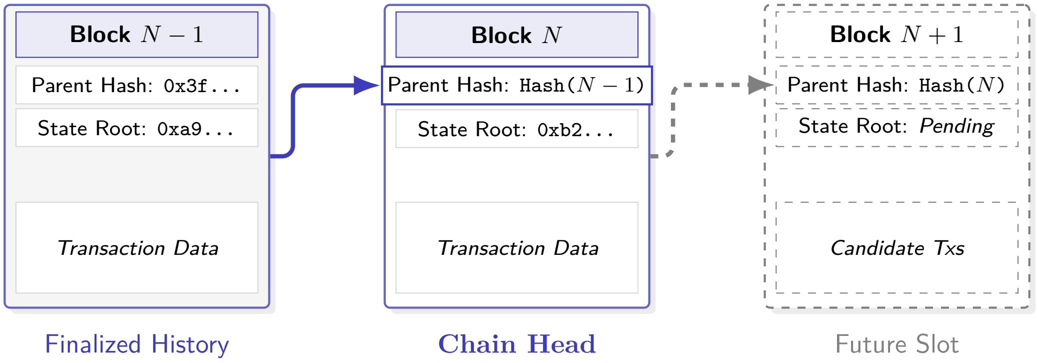


Figure 1.1: The cryptographic chain structure. Notice how the *Parent Hash* field of Block N explicitly commits to the entire content of Block $N-1$. Changing any byte in Block $N-1$ would change its hash, breaking the link to Block N.

1.2 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.

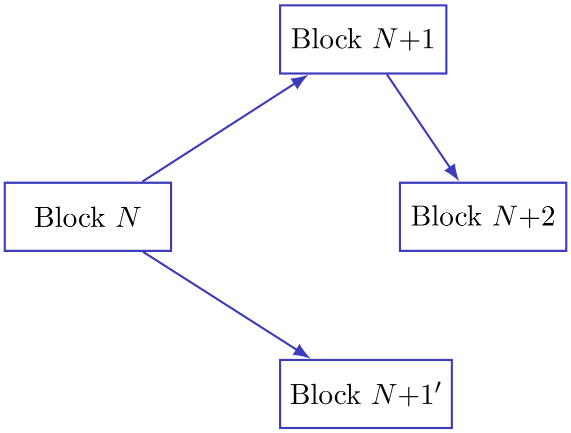


Figure 1.2: Temporary forks caused by network delays are resolved by the consensus rules, allowing the network to reconverge on a single chain.

1.2.2 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.

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.

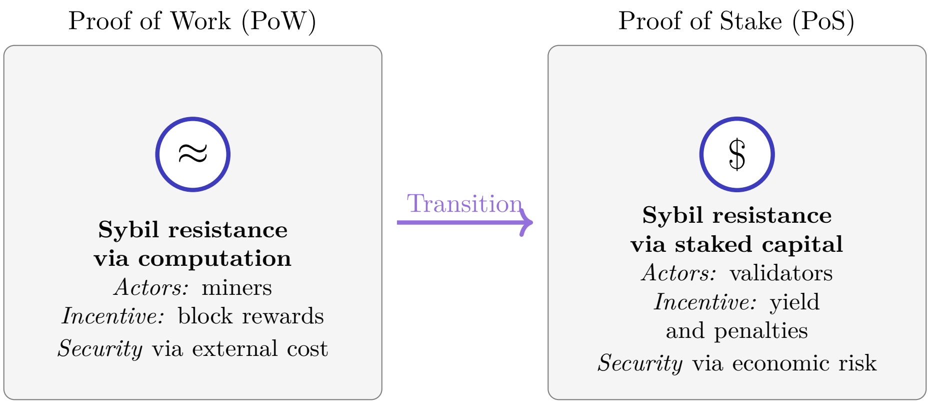


Figure 1.3: PoW and PoS use different Sybil-resistance resources. PoW relies on external expenditure; PoS relies on collateral that can be penalized by the protocol.

1.3 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 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 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.

Chapter 2

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 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 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.

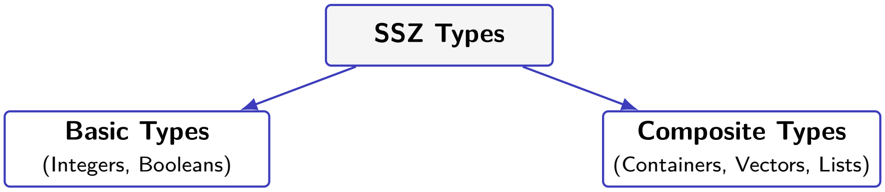


Figure 2.1: The SSZ type hierarchy distinguishes between fundamental values and structured data.

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 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.

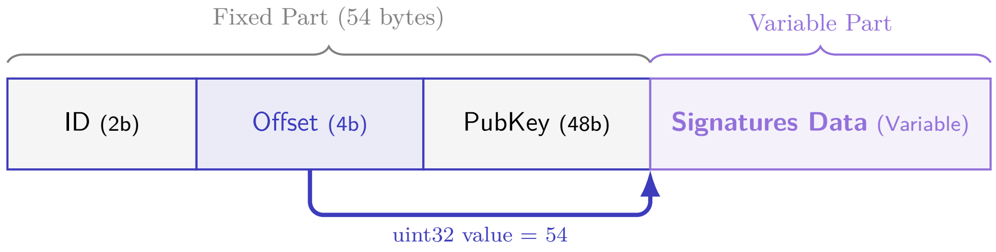
The offset represents the number of bytes from the start of the serialized object to the start of the specific variable-length data.

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.



The offset value (54) tells the deserializer to jump 54 bytes from the start to find the signature data. This preserves the fixed position of the PubKey field.

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.

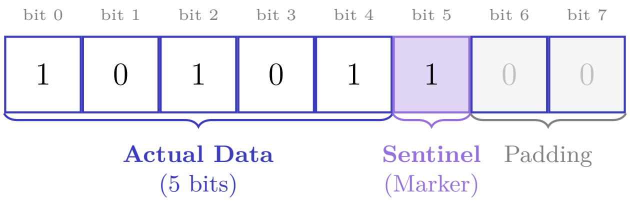


Figure 2.3: Bitlist Serialization. To preserve the exact length of the bit sequence within byte boundaries, a 1 (sentinel) is appended immediately after the data. The deserializer finds the highest-index 1 to determine the list length.

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.

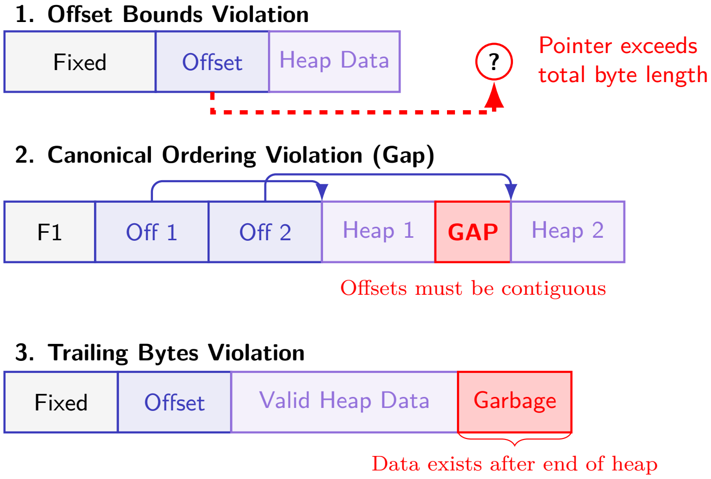

1. Offset Bounds Violation
Fixed
Offset
Heap Data
?
Pointer exceeds
total byte length
2. Canonical Ordering Violation (Gap)
F1
Off 1
Off 2
Heap 1
GAP
Heap 2
Offsets must be contiguous
3. Trailing Bytes Violation
Fixed
Offset
Valid Heap Data
Garbage
Data exists after end of heap

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 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.

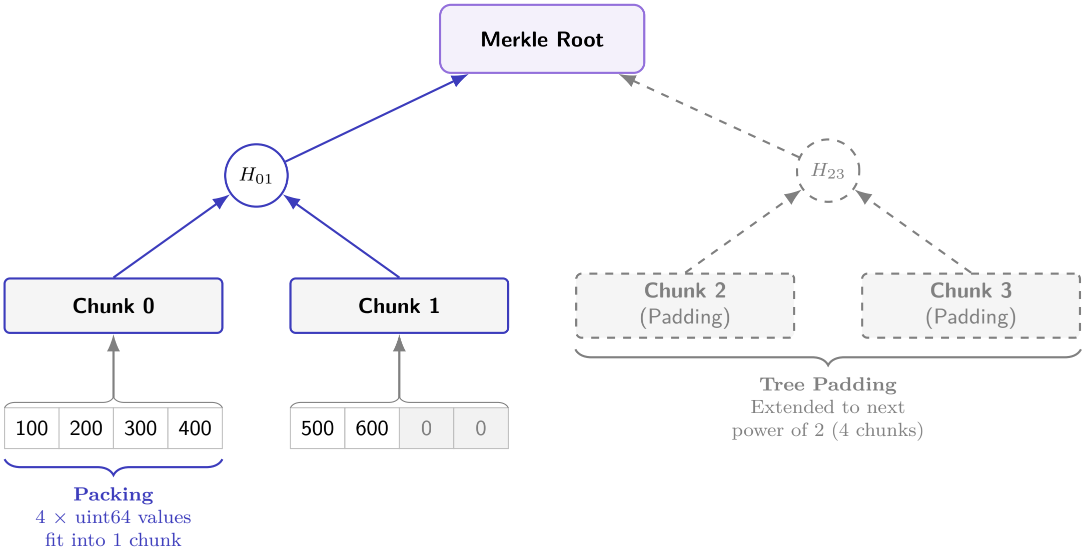


Figure 2.5: Packing and Merkleization. A list of six integers is packed into two chunks. The tree is then padded with zero-chunks to the next power of two (4 chunks) before hashing.

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.

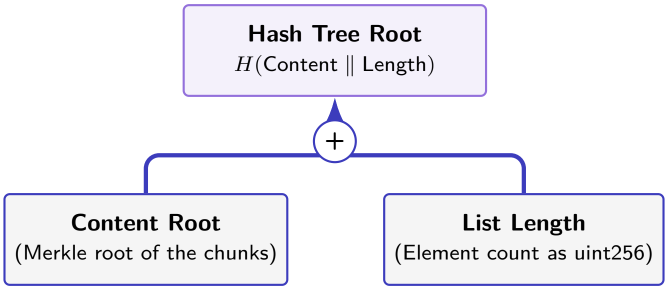


Figure 2.6: The Mix-in Length mechanism. The final identity of a List is a composite of its data and its size, preventing length-extension ambiguity.

2.5 Generalized Indices and Proofs

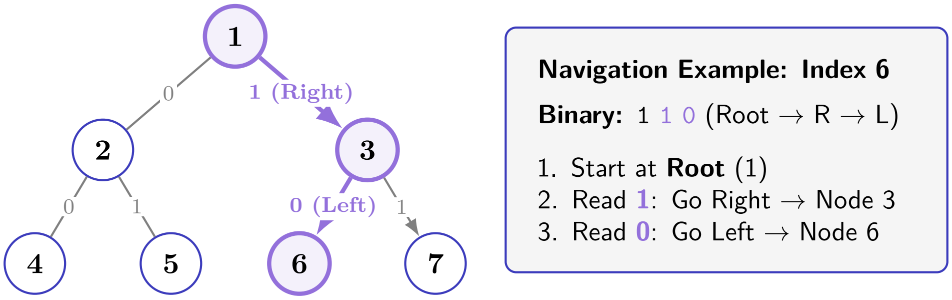
Generalized indices provide a deterministic addressing scheme for every piece of data within an SSZ structure. By mapping the data structure to a binary Merkle tree, we can assign a unique integer index to every node, from the root down to the individual leaves.

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).



Navigation Example: Index 6

Binary: 1 1 0 (Root → R → L)

1. Start at **Root** (1)
2. Read **1**: Go Right → Node 3
3. Read **0**: Go Left → Node 6

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)

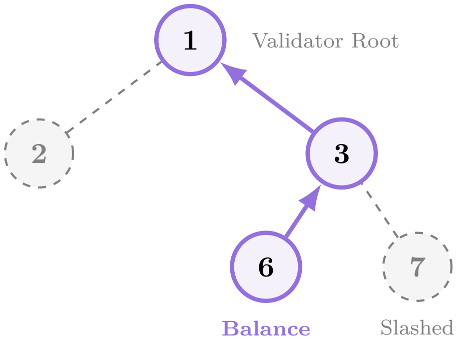
Since the container holds exactly four fields, these elements form the leaves of a binary tree with a depth of 2 (because $2^2 = 4$).

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).



Verification Logic

To prove **Balance (6)** is part of **Root (1)**:

1. Hash(**Balance**, Slashed) → **Node 3**
2. Hash(Node 2, **Node 3**) → **Root**

Gray nodes are provided in the proof.
Purple nodes are computed.

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.

Chapter 3

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 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.

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 Slot duration and intervals

Lean Consensus fixes two public parameters:

- Slot duration: $T_s = 4$ seconds.

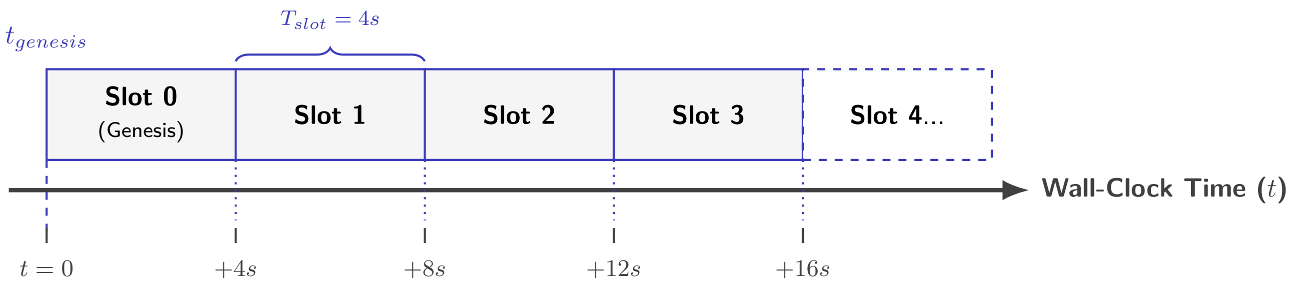


Figure 3.1: The relationship between physical wall-clock time and protocol slots. The genesis time $t_{genesis}$ anchors the system. Every 4 seconds, the protocol deterministically transitions to the next slot number.

- 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.

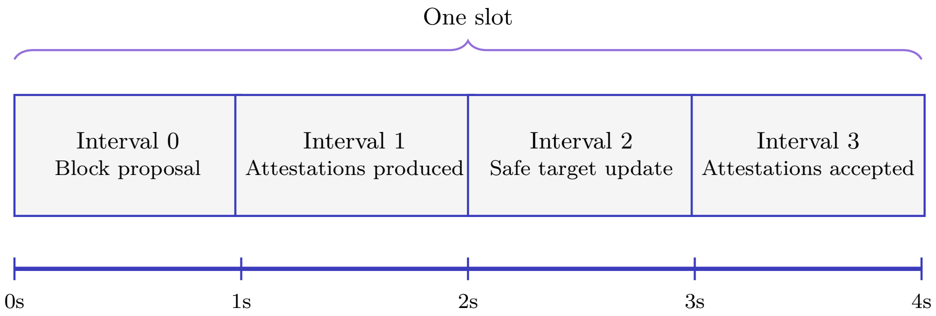


Figure 3.2: A slot is subdivided into four one-second intervals. Intervals define when different protocol actions occur.

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.

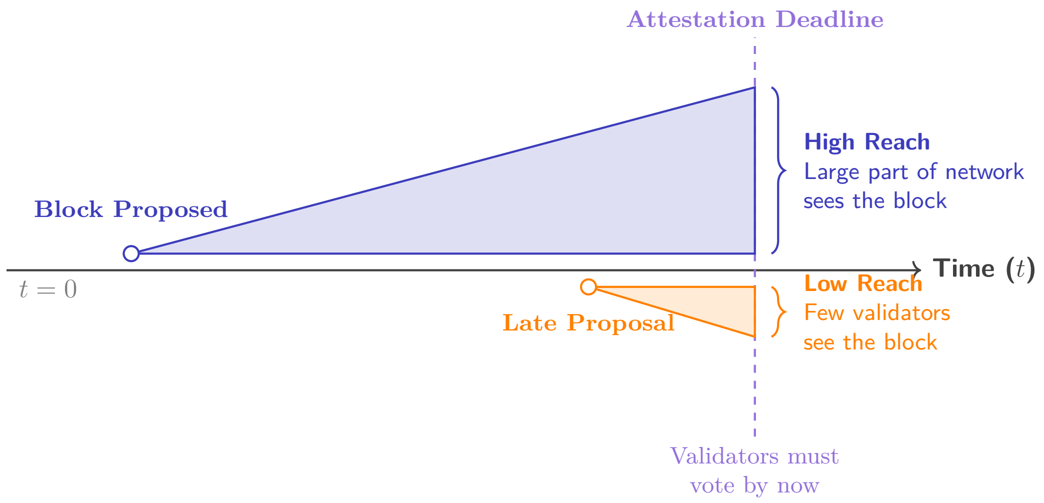


Figure 3.3: The race against the deadline. The vertical axis represents the percentage of the network that has received the block. An early proposal (blue) allows the gossip protocol to reach nearly all validators before the deadline. A late proposal (orange) leaves insufficient time for propagation,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

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

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:

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

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

3.3.1 Slot zero: the genesis slot

Slot 0 is defined as the genesis slot. It is the unique starting point of protocol time and the unique starting point of chain history. Unlike slots $1, 2, \ldots$, Slot 0 does not involve block production: it has no proposer assignment and no parent block reference.

Instead, the protocol begins from a genesis state that is constructed directly from the genesis configuration. This state is treated as the initial checkpoint for consensus. In particular, it is considered justified and finalized by definition, so fork choice has a fixed, non-revertible root from which to evaluate all later blocks.

When the first block is processed (in Slot 1 or later), it is linked to this genesis state. From that point on, every block's ancestry ultimately traces back to the Slot 0 genesis checkpoint.

3.4 Genesis configuration

Before the protocol can operate, all nodes must agree on an identical initial state. This state, denoted S_0, is derived from a fixed set of parameters called the genesis configuration. The purpose

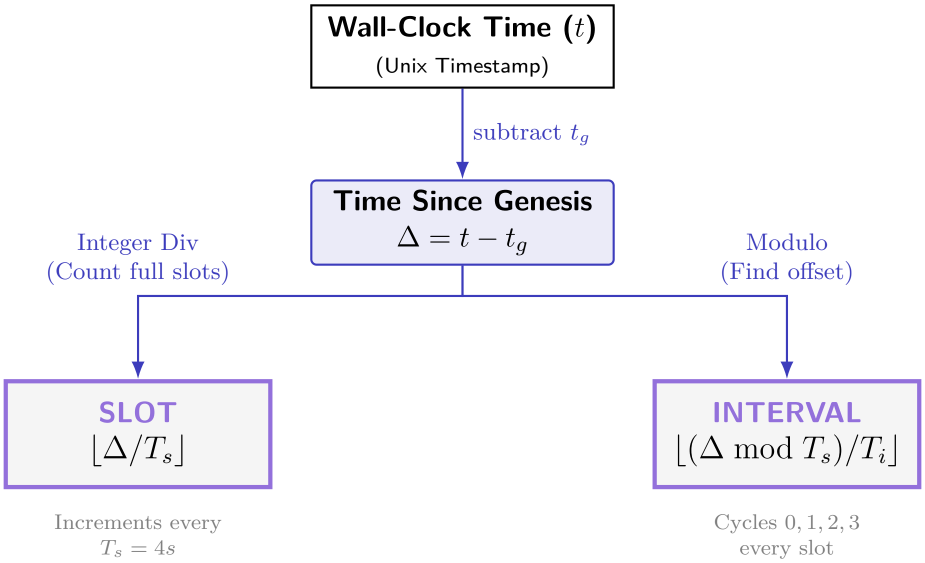


Figure 3.4: The slot clock logic. Wall-clock time is first normalized relative to genesis, then split into two coordinates: the Slot Number (coarse time) and the Interval (fine-grained offset within the slot).

of the genesis configuration is to allow independent nodes to start from the same state without exchanging messages or observing the network.

The genesis configuration provides exactly the information required to define protocol time and the initial validator set:

- Genesis time (t_g): a Unix timestamp, in seconds, that defines the start of Slot 0. All slot and interval computations are derived from this value. If nodes disagree on t_g, they will compute different slot numbers for the same wall-clock time and will therefore be unable to coordinate protocol actions.
- Genesis validators: a fixed list of validator public keys (XMSS keys). These keys define the initial validator registry and the initial distribution of stake. Each validator is assigned a unique integer index, starting from 0, which is used throughout the protocol to identify proposers and attesters.

Figure 3.5 shows how this configuration is transformed into the initial protocol state.

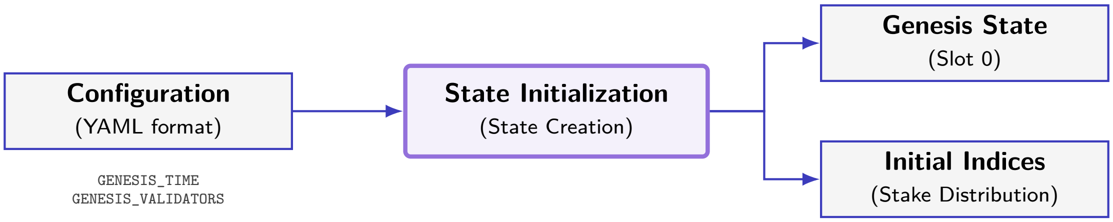


Figure 3.5: The initialization pipeline. The configuration is parsed to generate the immutable Slot 0 state, allowing for deterministic index assignment and stake distribution.

3.4.1 Initialization and state derivation

To initialize the chain, nodes initialize the state, which maps the configuration data into the protocol's state transition machine. This process is strictly deterministic:

1. **Validator Indexing:** Public keys are assigned sequential integer indices. These indices are used throughout the protocol's lifecycle to identify proposers and attesters.
2. **State Finalization:** The resulting genesis state is treated as the first finalized checkpoint. This allows the fork choice rule to have a valid, non-revertible root from which to begin evaluating new blocks.
3. **Slot 1 Trigger:** Once the current wall-clock time t reaches $t_g + T_s$, the protocol transitions to Slot 1. The validator assigned to the first index in the proposer schedule then broadcasts the first block, referencing the genesis state as its parent.

By standardizing this configuration in a cross-client format, Lean Consensus ensures that nodes written in different languages—but using the same parameters—converge on the same canonical head from the first active slot.

3.5 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.

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].

3.6 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.

Chapter 4

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 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.

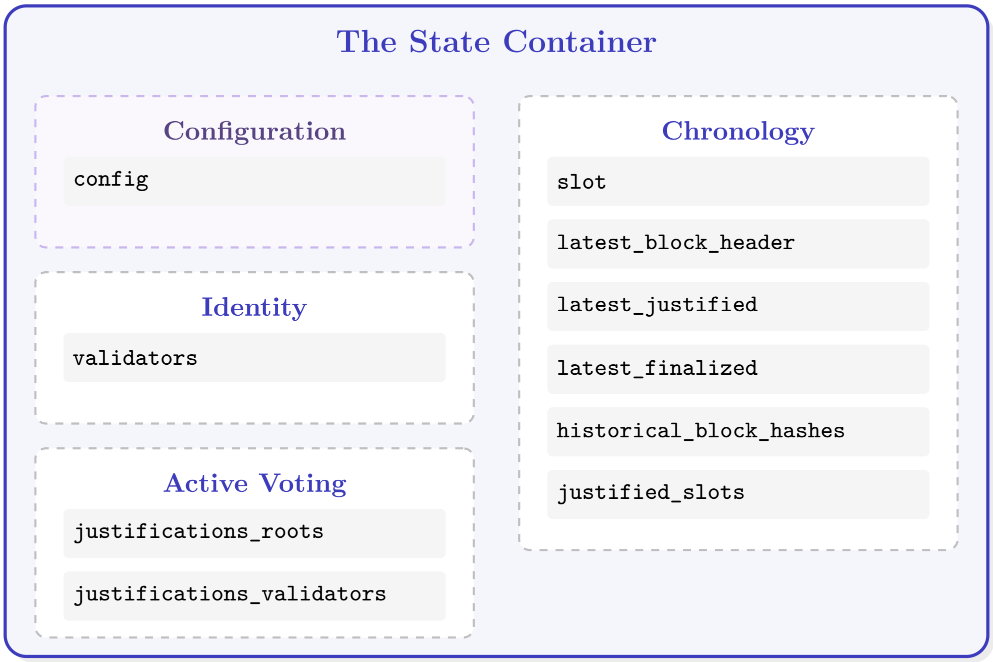


Figure 4.1: The Architecture of the State. The state object aggregates four distinct types of data: static configuration, the validator registry, historical context, and the active consensus scratchpad.

4.2 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.

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.

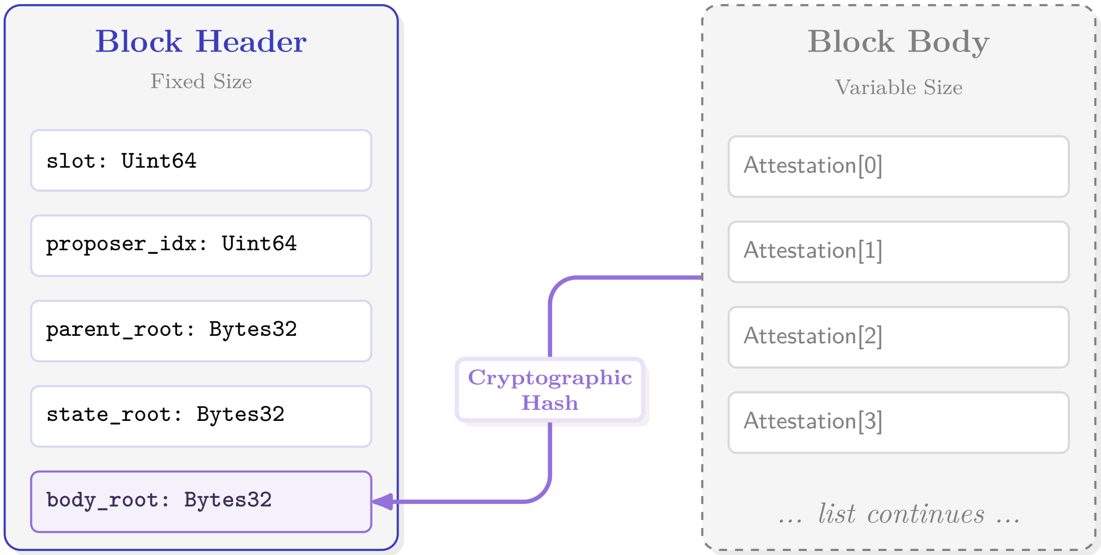


Figure 4.2: The cryptographic commitment. The header contains metadata and the body root, which is a hash of the variable-length body. This allows the header to serve as a constant-size proxy for the full block data.

4.3 The Transition Pipeline

The state transition function is applied as a deterministic validation pipeline that every honest node must execute in exactly the same way for every incoming block. This pipeline defines the precise sequence of operations that transform the current state into the next state. Determinism is crucial: if nodes were to apply different procedures or process blocks in a different order, the network would diverge into inconsistent versions of the chain.

Before a block can be accepted, the verifier must establish four properties. The local state must first be synchronized with the block's slot, since blocks may arrive after one or more empty slots. The block must then be structurally valid and authorized to extend the existing chain. Next, the consensus operations contained in the block body must produce legitimate updates to voting and checkpoint status. Finally, the resulting state must match the cryptographic commitment declared by the proposer in the block header.

These requirements define four sequential phases of the transition pipeline:

- **Time Synchronization:** The state is advanced through any skipped slots until it reaches the slot of the incoming block, ensuring that execution begins from the correct point in protocol time.
- **Structural Validation:** The block header is checked for correctness, including proposer authorization, parent linkage, and chronological consistency with the current chain.
- **Consensus Execution:** The block body is processed, applying attestations and updating justification and finalization according to the consensus rules.
- **Integrity Verification:** The new state is hashed and compared against the state root declared in the block header, confirming that the proposer's claimed transition matches the verifier's computation.

Only if all four phases succeed is the block accepted and the state updated; otherwise, the block is rejected and the state remains unchanged. The following subsections describe each phase in detail.

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.

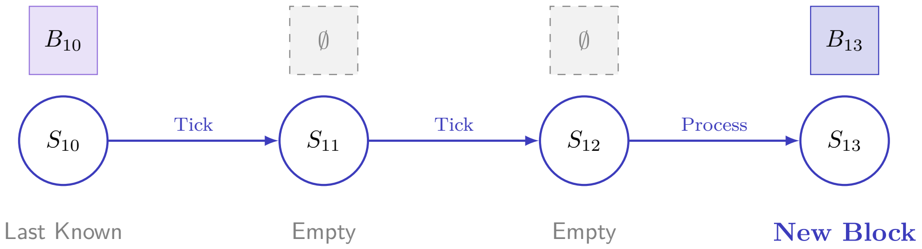


Figure 4.3: Processing empty slots. The state must conceptually tick through empty slots 11 and 12 before block 13 can be applied.

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.

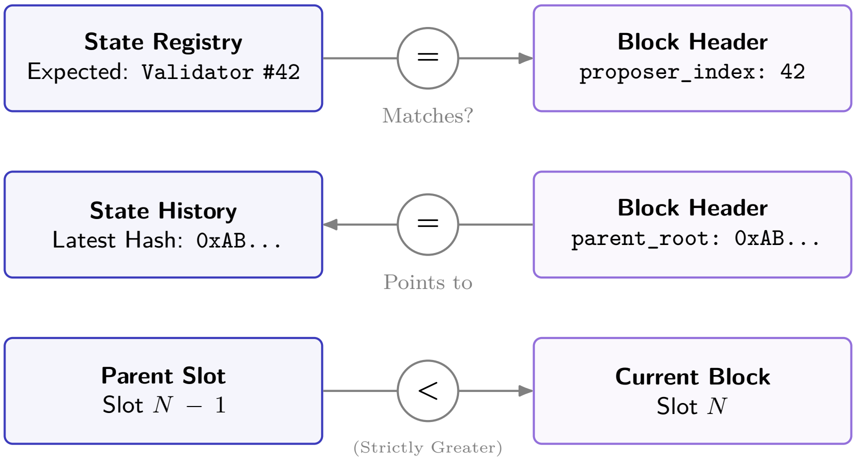


Figure 4.4: The three structural checks of the header validation phase. The incoming block (purple) is compared against the local state (blue) to ensure authorization, continuity, and chronological order.

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:

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

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

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

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.

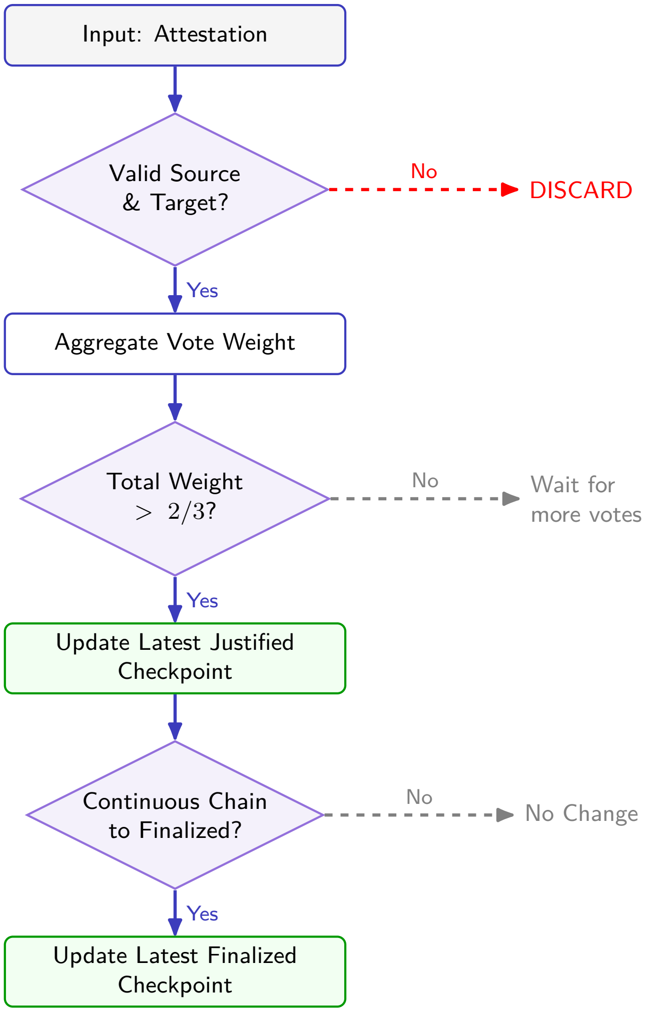


Figure 4.5: The consensus logic flow. For every attestation in the block, the system validates it, tallies the weight, and checks if it triggers a justification or finalization update.

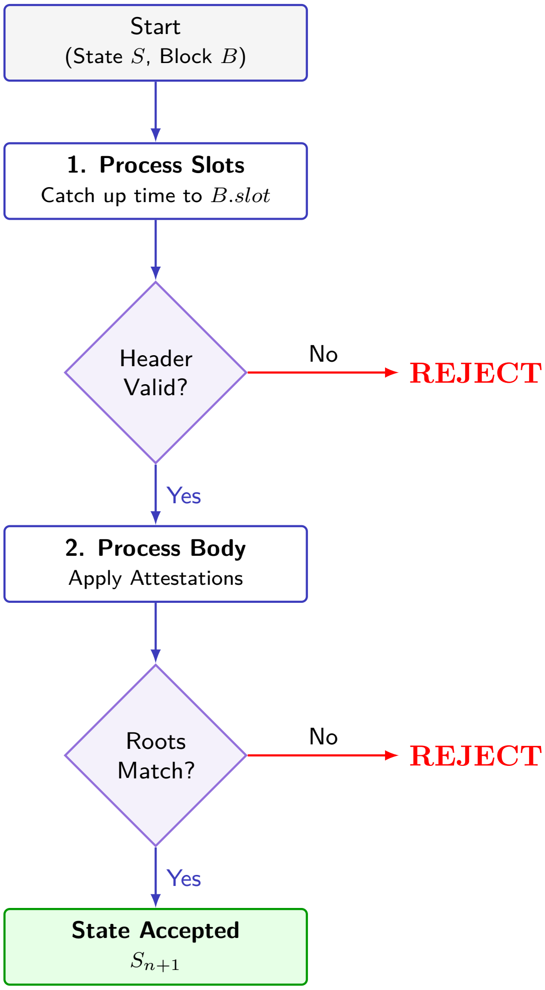


Figure 4.6: The block verification pipeline. Validation ensures that only blocks satisfying all protocol rules are applied to the local state.

Chapter 5

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 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.

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

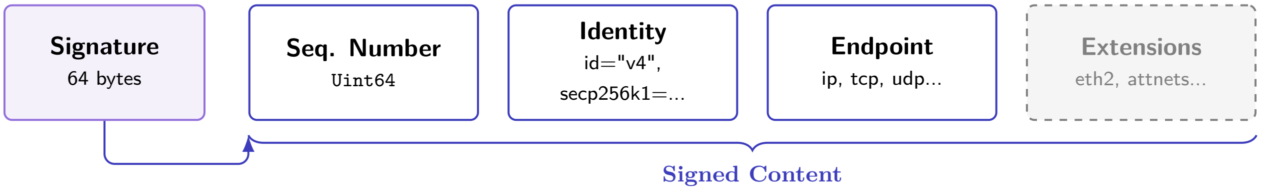


Figure 5.1: The ENR structure. The signature covers the sequence number and all key-value pairs. The key-value pairs provide flexibility, allowing the record to store generic network info (IP/UDP) alongside consensus-specific data.

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.

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.

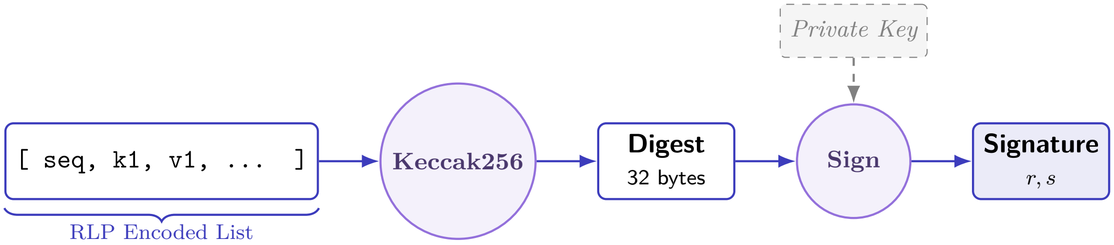


Figure 5.2: The ENR signing pipeline. The content list is RLP-encoded and hashed to create a fingerprint, which is then signed using the node's private key.

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.

5.1.3.1 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.

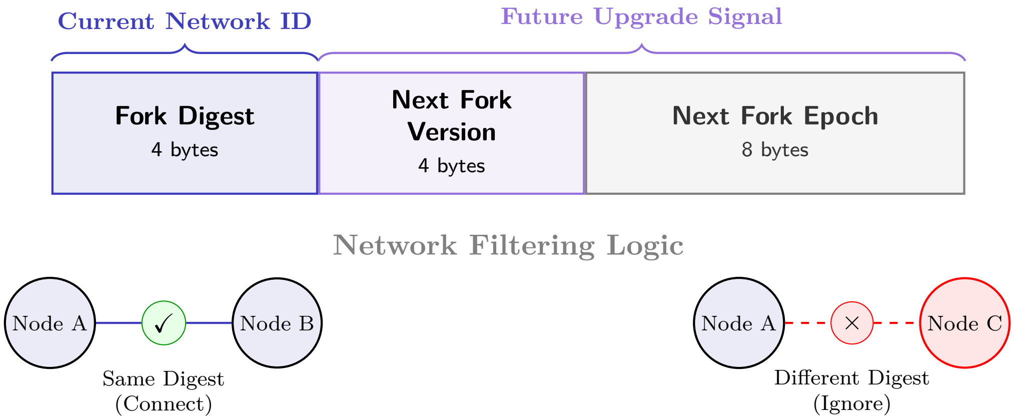


Figure 5.3: The `eth2` key structure. The first 4 bytes (Fork Digest) act as a strict firewall: if this value does not match, nodes simply ignore each other. The remaining 12 bytes allow nodes to coordinate future upgrades.

5.1.3.2 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 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.

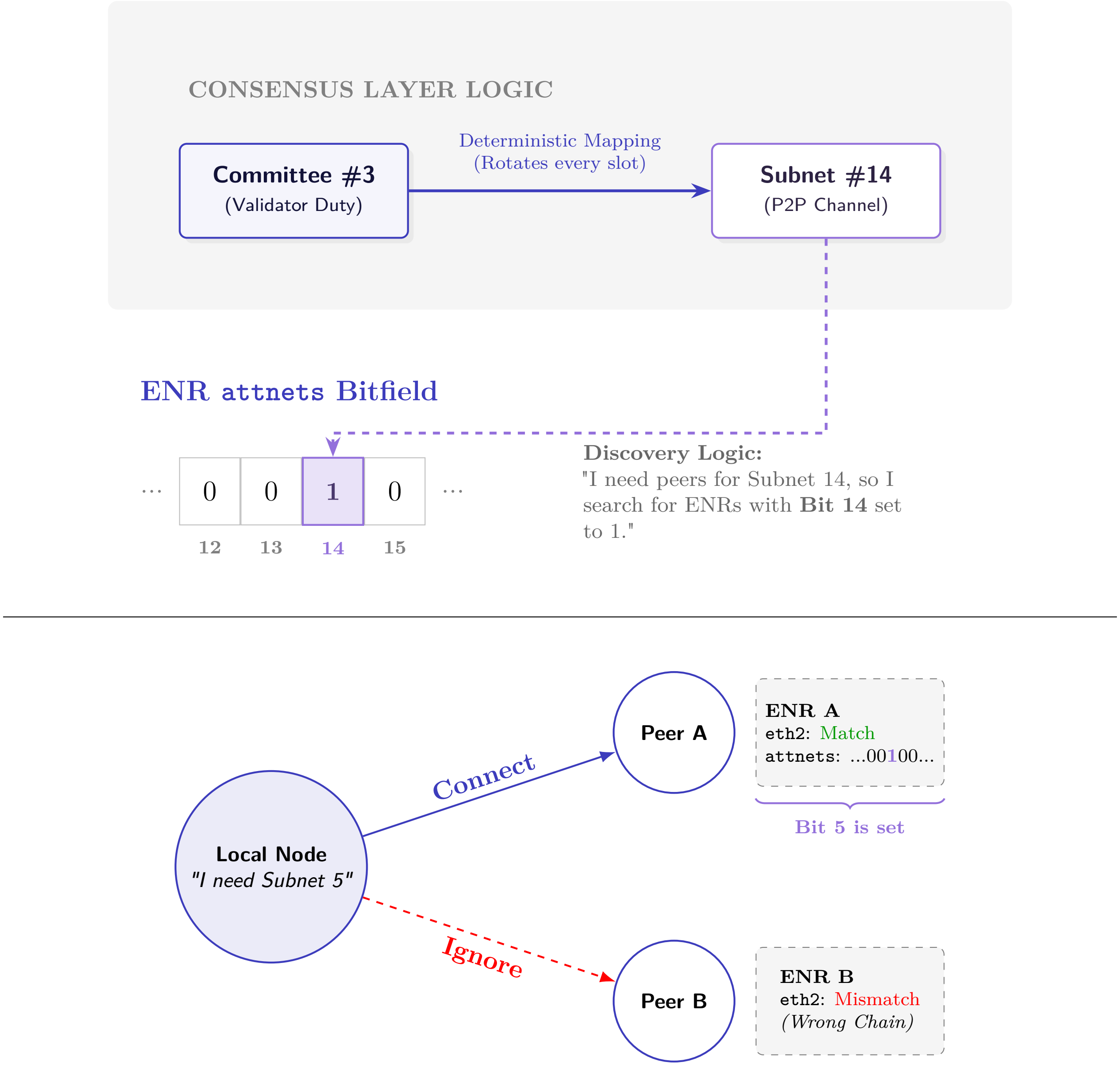
2. **The Transport (Subnets):** To facilitate this, the P2P network is sliced into 64 separate data channels called subnets (subscription networks). Think of a subnet as a dedicated radio frequency.

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.



Discovery Logic:
"I need peers for Subnet 14, so I search for ENRs with **Bit 14** set to 1."

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.

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:-IS4QHCYrYZbAKWCBRlAy5zzaDZXJBGkcnh4MHcBFZntXNFrdvJjX04jRzjzCBOonrkTfj499SZuOh8R33Ls8RRcy
5wBgmlkgnY0gmlwhH8AAAGJc2VjcDI1NmsxoQPKY0yuDUmstAHYpMa2_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.

5.2 Discovery v5: The Node Directory

While the Ethereum Node Record (ENR) provides a secure format for identity, it does not solve the problem of network entry. A new node joining the consensus layer initially knows only a small set of hardcoded bootnodes. To participate in block propagation and attestation, it must rapidly expand this list into a comprehensive view of the active network.

This bootstrapping process is handled by the Node Discovery Protocol version 5 (Discv5). Unlike the high-level application protocols which run over connection-oriented transports like TCP or QUIC, discovery operates entirely over UDP. This design choice makes the protocol lightweight and resilient, allowing it to maintain a massive, churn-heavy directory of peers without the overhead of establishing persistent connections.

Conceptually, Discv5 implements a Distributed Hash Table (DHT) based on the Kademlia algorithm. In this model, the global directory is sharded across the network itself. Every node maintains a routing table containing the ENRs of the peers mathematically closest to it. By querying these nodes, any participant can progressively hop through the network to locate any specific target—whether that target is a single peer or a subnet of validators.

5.2.1 The XOR Distance Metric

In a physical network, the distance between two computers is measured by latency: the time it takes for a packet to travel from point A to point B. However, Ethereum is a decentralized overlay network that must remain robust even if specific physical links fail. To achieve this, Discv5 defines distance logically rather than physically, based entirely on the similarity of Node IDs.

Since a Node ID is simply a 256-bit number, the protocol calculates the distance between two nodes, A and B, using the XOR (exclusive OR) operation:

$$\text{Distance}(A, B) = \text{ID}_A \oplus \text{ID}_B$$

The resulting value is treated as a 256-bit integer. If two IDs are identical, the result is zero. As the IDs become more distinct in their binary representation, the integer value of the XOR result increases, representing a greater mathematical distance.

5.2.1.1 Greedy Routing and Convergence

To navigate the global network, the protocol employs a *greedy routing* algorithm. The fundamental goal of this algorithm is to find a specific Target ID.

In the context of the DHT, the Target ID represents a precise location in the 256-bit keyspace. This location is significant because of the protocol's storage logic: information about a node or a topic is always stored on the peers that are numerically closest to it. Therefore, to retrieve information, a node must route its query to the neighborhood of that Target ID.

When a node initiates a search, it does not broadcast the request blindly. Instead, it uses a greedy heuristic: *always choose the peer that reduces the distance to the target.* As illustrated in Figure 5.5, consider a scenario where a Start Node (A) needs to find a Target Node (T). The routing logic proceeds in three steps:

1. **Measurement:** Node A calculates its own distance to the target (d_A). This distance defines a search radius, depicted as the circular boundary in the figure.
2. **Evaluation:** Node A scans its connected peers to see if any fall within this radius.
   - **Peer C** is physically nearby, but its XOR distance to the target is larger than Node A's ($d_C > d_A$). It falls outside the search radius (the white region) and is ignored.
   - **Peer B** is located inside the search radius (the shaded blue region). Its distance to the target is strictly smaller than Node A's ($d_B < d_A$).

3. **Selection:** Node A selects Peer B as the next hop. By forwarding the query to B, the search radius effectively shrinks.

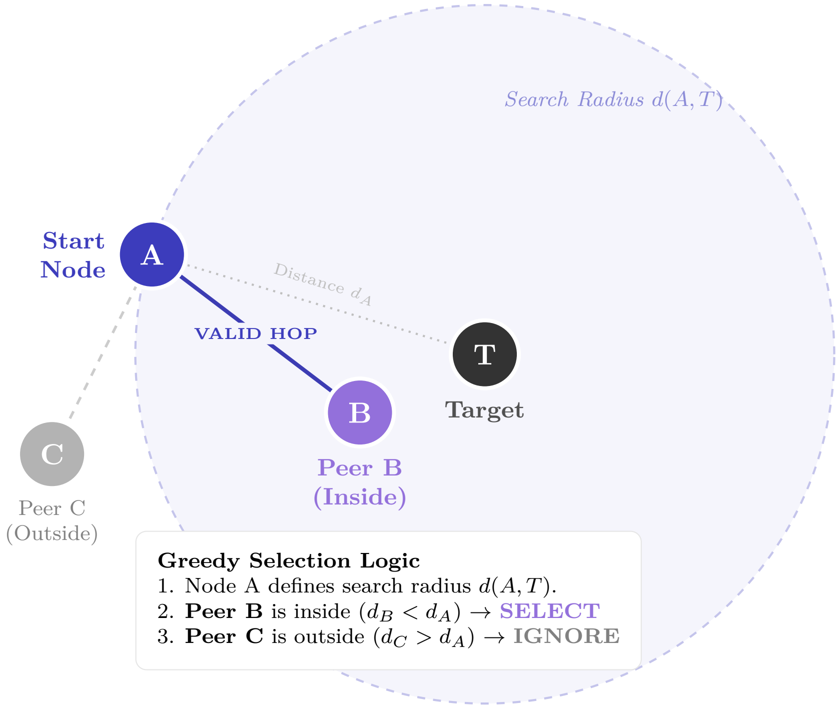
Because the distance metric is an integer that strictly decreases with every successful hop, the algorithm is mathematically guaranteed to converge on the target, preventing infinite routing loops.

5.2.1.2 The Triangle Inequality

The validity of this greedy approach relies on the XOR metric satisfying the triangle inequality:

$$\text{Distance}(A, T) \leq \text{Distance}(A, B) + \text{Distance}(B, T)$$

This inequality ensures that the logical topology is consistent. It guarantees that if Peer B is numerically closer to the target, it is genuinely better positioned in the overlay network. This prevents logical wormholes where a node might appear close locally but be topologically distant, ensuring that every greedy decision translates into global progress.



Greedy Selection Logic
1. Node A defines search radius $d(A, T)$.
2. **Peer B** is inside ($d_B < d_A$) → **SELECT**
3. **Peer C** is outside ($d_C > d_A$) → **IGNORE**

Figure 5.5: Geometric Visualization of Greedy Routing. The dashed circle represents the current distance to the target. To make progress, Node A must find a peer strictly inside this circle (the shaded region). Peer C is ignored because it lies outside the boundary, meaning a hop to C would increase the distance to the target.

5.2.2 The Routing Table

While the XOR operation provides precise ordering, working with raw 256-bit integers is computationally unwieldy for routing logic. To simplify this, the protocol groups nodes based on their *logarithmic distance*.

This metric effectively measures the length of the shared prefix between two IDs. When two 256-bit numbers are XORed, the position of the most significant bit (the first ‘1‘) determines the magnitude of the distance. If the result has many leading zeros, it means the IDs shared a long sequence of identical bits before diverging.

For example, if two Node IDs share the first 250 bits, the XOR result will be a very small number, and they are considered neighbors. If they differ at the very first bit, the result is a massive number, and they are considered far apart. The routing table uses this to group peers into 256 buckets: nodes that share a prefix of length N are placed into bucket $256 - N$. This allows the node to maintain high-fidelity knowledge of its closest mathematical neighbors while keeping only a sparse, logarithmic view of the rest of the network.

As illustrated in Figure 5.6, this logarithmic structure creates a specific view of the network: the node maintains high-fidelity knowledge of its closest mathematical neighbors (those sharing long prefixes) while keeping only a sparse, representative sample of distant nodes. This allows any node to locate any other node in the global network in $O(\log n)$ hops, balancing memory efficiency with routing speed.

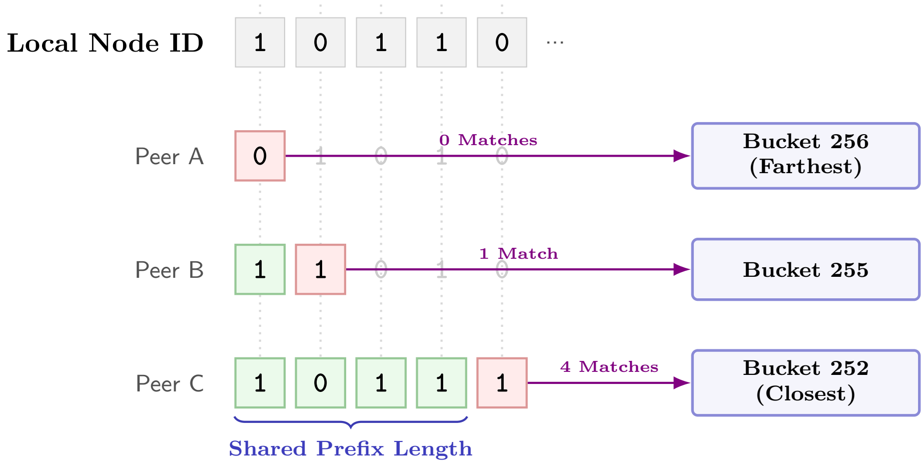


Figure 5.6: Logarithmic Bucketing logic. The routing table groups peers based on the length of their shared prefix with the local node. A single mismatch at the start (Peer A) places a node in the farthest bucket. A long sequence of matching bits (Peer C) places a node in a much closer bucket.

5.2.2.1 Metric Precision vs. Topological Grouping

It is important to distinguish between the organization of the routing table and the mechanism used for traversal. While k-buckets classify peers into equivalence classes based on shared prefix lengths, this grouping alone is insufficient for efficient routing.

Within a single bucket, all peers are topologically equivalent regarding the local node. However, relative to a remote target, they may differ significantly in distance. The XOR metric provides a strict total ordering over the keyspace. This allows the routing algorithm to discriminate between peers within the same bucket, ensuring that the selected next hop satisfies the strict descent property $d(\text{next}, \text{target}) < d(\text{current}, \text{target})$. Without this metric precision, the greedy routing algorithm could not guarantee monotonic convergence toward the destination.

5.2.3 The Wire Protocol: Obfuscation and Security

While Kademlia defines the logic of discovery, the Wire Protocol defines the secure conversation between nodes. A critical design requirement for Ethereum is censorship resistance. In many restrictive environments—ranging from corporate intranets to nation-states with strict internet controls—administrators actively attempt to identify and suppress peer-to-peer traffic.

To achieve this, firewalls often employ Deep Packet Inspection (DPI). These tools scan the raw data of network packets looking for static, predictable patterns known as magic bytes. For example, if every Discovery packet began with a fixed header like `0xETH`, a firewall could easily be configured to detect that signature and drop the connection immediately. To survive in these hostile environments, Discv5 is designed to be indistinguishable from random noise. It eliminates all plaintext headers and static identifiers, ensuring that a passive observer cannot determine that a packet belongs to the Ethereum protocol.

5.2.3.1 Packet Anatomy

To achieve this indistinguishability while ensuring the intended recipient can still decode the message, the packet is divided into three distinct sections, as illustrated in Figure 5.7.

1. **The Masking IV (16 bytes):** Every packet begins with a random 16-byte Initialization Vector (IV). This serves two purposes. First, it ensures that two identical messages sent to the same node result in completely different ciphertexts. Second, it acts as the entropy source for the header masking.

2. **The Masked Header:** This section contains the protocol metadata, such as the packet type (e.g., `PING`) and the request ID.

   Crucially, this header is *not* encrypted with a session key, because the node might not have a session yet. Instead, it is masked using a key derived from the destination's Node ID. This relies on a simple premise: to send a message to a node, you must already know its ID. Therefore, you can use that ID to obfuscate the header so that only the intended recipient (who knows their own ID) can unmask it. To any other observer, this section looks like garbage bytes.

3. **The Encrypted Message:** The final section contains the actual payload (for example, a list of ENRs in a `NODES` response). This data is fully authenticated and encrypted using an AES-GCM session key. This key is established during the handshake process and ensures that the content remains confidential and tamper-proof.

5.2.4 The Lazy Handshake

Unlike traditional transport protocols like TCP, which require a formal three-way handshake before any data can be exchanged, Discovery v5 is designed for optimistic communication. A node does not ask permission to speak; it simply speaks.

When an initiator (Node A) wishes to query a recipient (Node B), it assumes a session might already exist. It encrypts the query—for example, a `FINDNODE` command, see Figure 5.8—using the session keys it has stored in memory. If no session exists, or if the keys are stale, the node generates

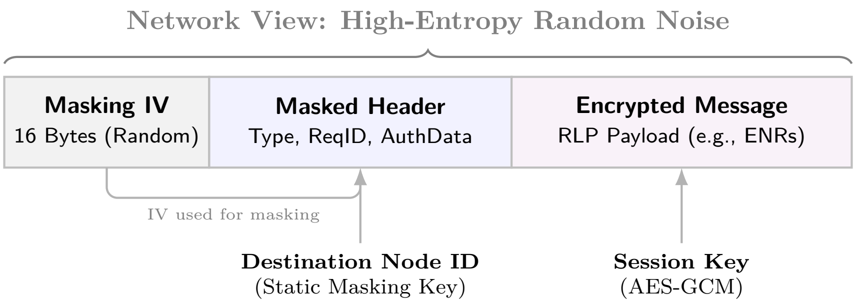


Figure 5.7: The Discv5 Packet Structure. The packet is designed to hide protocol metadata from firewalls. The header is masked using the receiver's ID (allowing decryption without a session), while the content payload is fully encrypted using negotiated session keys.

a random key and encrypts the packet with it. To the network, this looks like a standard encrypted message.

When Node B receives this packet, it attempts to decrypt it. If the decryption fails—which it will if the keys are random or outdated—Node B does not silently drop the packet. Instead, it recognizes this as an implicit connection attempt and responds with a challenge. This triggers a verification workflow known as the *Lazy Handshake*. This mechanism ensures that expensive cryptographic handshakes only occur when strictly necessary, rather than for every new connection.

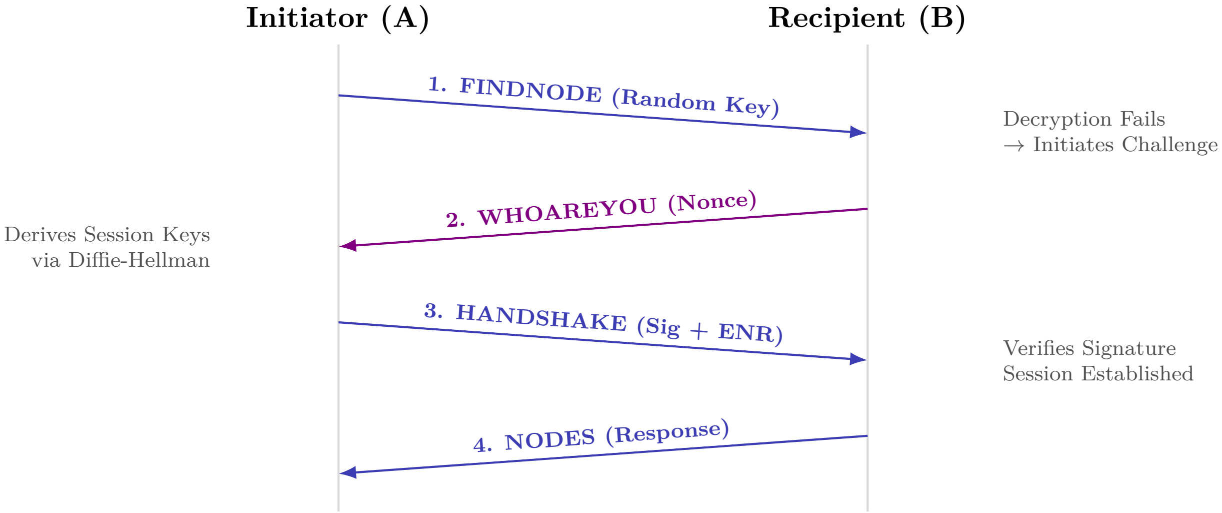


Figure 5.8: Sequence Diagram of the Lazy Handshake. The communication begins optimistically with a standard message. If the recipient cannot decrypt it (because no session exists), it issues a `WHOAREYOU` challenge. The initiator must then prove its identity by signing the nonce, which establishes the shared session keys for subsequent messages.

5.2.4.1 Responding to the Challenge

The `WHOAREYOU` packet is a demand for proof. It serves to verify two critical properties: *liveness* (confirming the initiator is actually reachable at the claimed IP) and *identity* (confirming the initiator controls the private key linked to their Node ID).

To satisfy this demand and establish a secure session, the node performs a specific cryptographic sequence. This process is designed to guarantee Forward Secrecy and prevent replay attacks. The sequence proceeds as follows:

1. **Ephemeral Key Generation:** First, the node generates a fresh, temporary keypair known as the *ephemeral key*. This key is used exclusively for this single session. By using a disposable key for encryption, the protocol ensures that even if the node's long-term private key is stolen in the future, attackers cannot decrypt past conversations (Forward Secrecy).

2. **Challenge Data Construction:** To prevent replay attacks, the node must cryptographically bind its response to the specific challenge it just received. It constructs a byte sequence called the *challenge data*, which combines the packet's unique masking IV and the metadata from the `WHOAREYOU` packet. This ensures the response is valid only for this specific moment in time.

3. **Identity Proof (Signing):** The node signs this challenge data using its *long-term* identity private key (secp256k1). This signature is the core authentication step: by signing a package that includes the challenge details, the node proves to the recipient that it is the legitimate owner of the Node ID.

4. **Key Derivation (HKDF):** The node calculates the shared encryption secret using an Elliptic Curve Diffie-Hellman (ECDH) exchange. It mathematically combines its own *ephemeral* private key with the recipient's *static* public key (retrieved from the recipient's ENR).

   The resulting shared secret is passed through a Key Derivation Function (HKDF). This function splits the secret into two symmetric AES-GCM keys: an `initiator-key` (used to encrypt outgoing messages) and a `recipient-key` (used to decrypt incoming messages).

5. **Handshake Packet Assembly:** Finally, the node re-sends its original query. This time, the packet acts as a handshake: it includes an Authentication Header containing the signature, the ephemeral public key, and the node's ENR. The message payload is now fully encrypted using the newly derived session keys.

When the recipient receives this handshake packet, they verify the signature against the node's public ID. If valid, they perform the matching ECDH operation to derive the exact same session keys, allowing the secure conversation to proceed.

5.2.5 The Iterative Lookup

Once the secure transport is established, the node begins the actual work of discovery: the lookup. Whether the goal is to find a specific peer for the DHT or to discover a subnet for consensus duties, the process relies on the `FINDNODE` command.

While often referred to as a recursive lookup in theoretical literature, the implementation in Discv5 is strictly iterative. The initiating node controls the entire process, rather than handing off the responsibility to other peers. The algorithm, as illustrated in Figure 5.9, proceeds as follows:

1. **Selection (α Concurrency):** The node selects the α closest peers to the target from its local routing table. In Ethereum, the concurrency parameter is typically set to $\alpha = 3$. This parallelism prevents a single slow or non-responsive peer from stalling the search.

2. **The Query Phase:** The node sends a `FINDNODE(TargetID)` request to these α peers simultaneously. This request asks: "Please send me the k nodes you know that are closest to this Target ID."
3. **The Update Phase:** As responses (`NODES` packets) arrive, the initiator calculates the XOR distance of the newly discovered peers relative to the target.
    - If a new peer is closer than the ones already visited, it is added to the candidate list.
    - If a peer is farther, it is discarded.
4. **Convergence:** The loop repeats. The node selects the next set of α closest unvisited peers from its updated list and queries them. The search terminates when a round of queries fails to return any nodes closer than the ones already seen.

This mechanism allows an Ethereum node to zoom in on any target in the global mesh. By querying only a logarithmic number of peers, it can locate a specific node among millions within moments, maintaining a bandwidth-efficient view of the network.

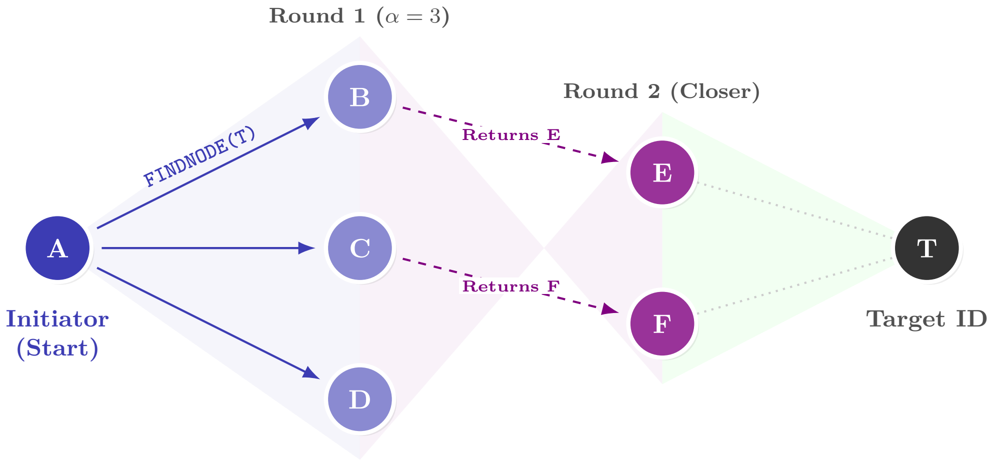


Figure 5.9: The Iterative Lookup Funnel. Node A searches for Target T. In Round 1, it queries the $\alpha = 3$ closest nodes it knows (B, C, D). These nodes return closer peers (E, F). In Round 2, A queries E and F, strictly reducing the distance to the target until convergence.

5.3 The Transport Layer

Discovery tells a node where other peers can be reached. The transport layer is what makes communication with those peers possible. It turns an IP address and port into a long-lived connection that can carry protocol messages in both directions.

The requirements are simple but strict. Messages must arrive reliably even though the underlying network drops packets, reorders them, and varies in latency. The connection must also regulate its sending rate so that a fast sender does not overwhelm a slow receiver or the network path between them. Finally, the transport must be secure: when a node connects to a peer, it must learn which node it is talking to, and the bytes exchanged on the connection must be confidential and integrity-protected.

Lean Consensus implements this using two concrete stacks. The default stack is QUIC over UDP, which provides stream multiplexing and encryption as part of the transport protocol. A TCP-based fallback exists for networks where UDP is blocked; in that case, encryption and authentication are provided by Noise on top of TCP. The following subsections describe these stacks and the security model they implement.

5.3.1 Protocol Stack Overview

The networking code in Lean Consensus is structured as a small stack of layers, each responsible for a specific task. Higher layers depend on lower ones, but do not need to know how they are implemented. This separation makes the system easier to reason about and allows individual layers to evolve without affecting the rest of the protocol.

From bottom to top, the stack consists of the following layers reprensented on Figure 5.10:

1. **Transport layer.** This layer is responsible for moving bytes between two machines on the internet. It deals with packet loss, reordering, congestion control, and connection management. In Lean Consensus, the primary transport is QUIC over UDP. TCP is supported as a fallback for networks where UDP traffic is unavailable or unreliable.

2. **Security layer.** Once a transport connection exists, the security layer authenticates the remote peer and encrypts all traffic on the connection. This ensures that messages cannot be read, modified, or forged by third parties. Lean Consensus uses TLS 1.3 when running over QUIC, and the Noise protocol framework when running over TCP.

3. **Multiplexing layer.** Consensus traffic is not uniform: some messages are small and latency-sensitive, while others may involve large data transfers. The multiplexing layer allows multiple independent streams of messages to share a single connection. This prevents slow or bulky traffic, such as state synchronization, from blocking time-critical messages like block or attestation gossip.

4. **Application layer.** This top layer contains the protocols that define *what* data is exchanged. Examples include gossip protocols for broadcasting new blocks and attestations, and request–response protocols for fetching specific data during synchronization.

5.3.2 QUIC: The Primary Transport

Before examining QUIC itself, it is useful to clarify what a transport protocol is expected to do. At the lowest level, the internet provides only a best-effort packet delivery service. Individual packets may be lost, duplicated, delayed, or delivered out of order. There is no guarantee that a packet sent by one machine will ever arrive at another. This service is intentionally minimal and places the burden of reliability on higher layers.

A transport protocol sits on top of this unreliable substrate and provides stronger guarantees to applications. For decades, the Transmission Control Protocol (TCP) has defined the standard model of reliable communication on the internet. In particular, TCP provides: reliable delivery (lost packets are detected and retransmitted), in-order delivery (data is delivered to the application in the same order it was sent), congestion control (the sending rate adapts to network conditions),

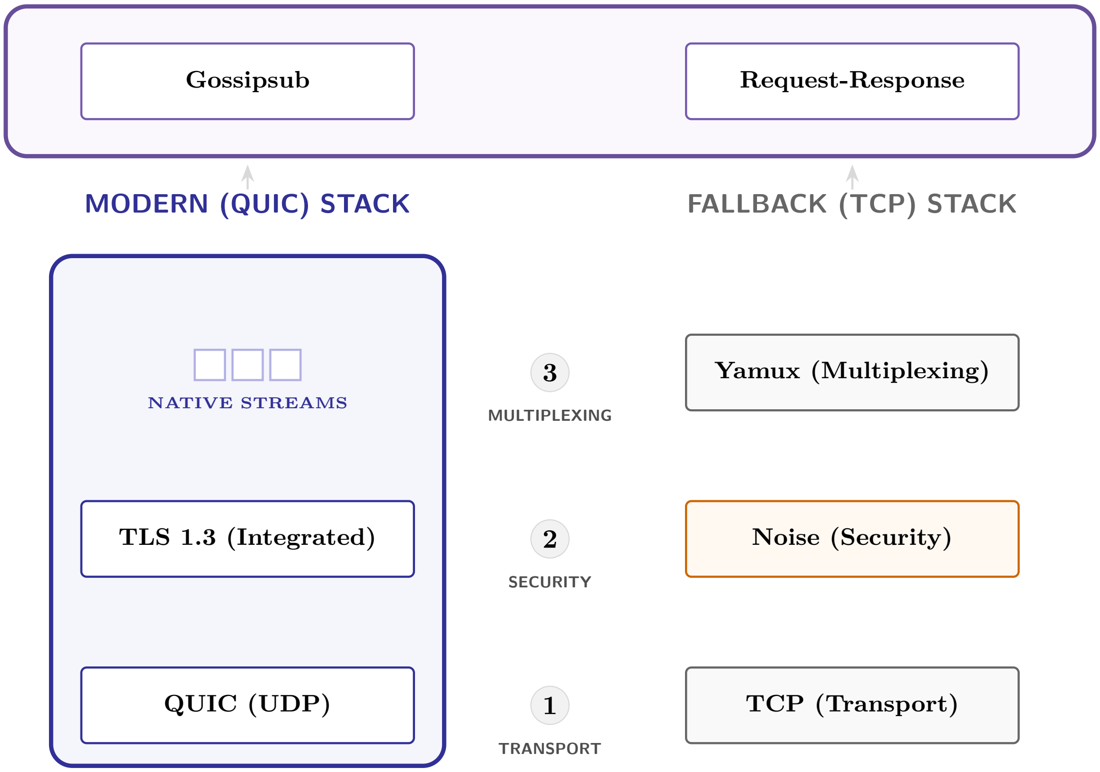


Figure 5.10: Standardized Transport Architecture. Every functional component across both the QUIC and TCP stacks utilizes a uniform protocol interface, ensuring predictable interaction with the consensus-layer sub-protocols.

and flow control (the receiver limits how much data it can accept). TCP exposes these guarantees through a single abstraction: a bidirectional, ordered byte stream. Applications write bytes into the stream on one side and read the same bytes, in order, on the other side.

Lean Consensus adopts QUIC, specified in RFC 9000 [8], because it provides the same fundamental guarantees as TCP while removing structural limitations that become problematic in modern peer-to-peer systems. QUIC is a connection-oriented transport protocol that runs over UDP, allowing it to be deployed across existing network infrastructure while implementing reliability, congestion control, and security entirely in user space.

The most important conceptual difference is that QUIC does not expose a single ordered byte stream. Instead, it provides a connection composed of multiple independent streams. This design choice has major consequences for latency, robustness, and protocol composition in consensus networks. The following subsections describe the key properties of QUIC that motivate its use in Lean Consensus.

5.3.2.1 Eliminating Head-of-Line Blocking

In TCP, all data sent over a connection belongs to a single ordered sequence of bytes. Even when an application logically multiplexes different kinds of traffic—such as block gossip, attestation gossip, and state synchronization—TCP treats all bytes as part of one stream. If a packet containing some bytes is lost, TCP must wait for that packet to be retransmitted and received before delivering any subsequent bytes to the application. This behavior preserves ordering but introduces an important performance problem.

When multiple logical streams share a TCP connection, a loss affecting one stream blocks progress for all others. Even if data for Stream B arrives successfully, it cannot be delivered while TCP is waiting for missing data from Stream A. This phenomenon is known as *head-of-line (HOL) blocking*. Figure 5.11 illustrates this effect.

QUIC avoids head-of-line blocking by making streams a first-class part of the transport protocol. A QUIC connection supports many independent streams, each of which is an ordered sequence of bytes with its own flow control and loss recovery state.

When a packet containing data for one stream is lost, only that stream is affected. Data belonging to other streams can continue to be delivered to the application as soon as it arrives. This property is particularly important for consensus systems, where small, latency-sensitive messages must not be delayed by unrelated traffic such as state synchronization or historical data transfer.

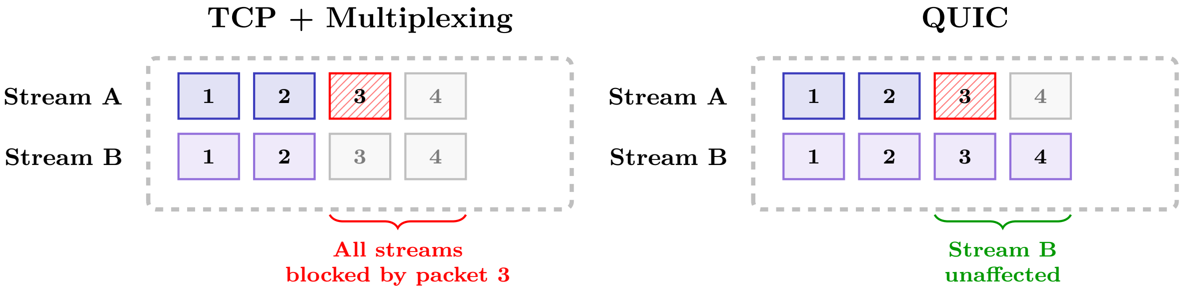


Figure 5.11: Head-of-line blocking comparison. QUIC isolates packet loss to the affected stream, preventing unrelated streams from stalling.

5.3.2.2 Low-Latency Connection Establishment (1-RTT)

Before application data can be exchanged securely, peers must establish a connection and derive shared cryptographic keys. In a traditional TCP+TLS stack, this process is layered. TCP first performs a three-way handshake to establish connectivity. Only after this completes does the TLS handshake begin, which negotiates encryption keys and authenticates the peer. This requires multiple round trips before encrypted application data can be sent, as illustrated in the legacy column of Figure 5.12.

QUIC integrates transport and security handshakes into a single protocol. Cryptographic keys and transport parameters are negotiated together using TLS 1.3, which is built directly into QUIC. In the common case, a secure connection is established in a single round trip (1-RTT).

If two peers have communicated before and cached cryptographic state, QUIC can send application data in the first flight of packets (0-RTT). While 0-RTT has replay-related limitations, it can significantly reduce connection latency for frequently interacting peers, reaching the theoretical minimum of receiving a data response in exactly one round trip.

For a peer-to-peer consensus network, reducing connection setup latency lowers the cost of churn and allows newly discovered peers to participate in gossip more quickly.

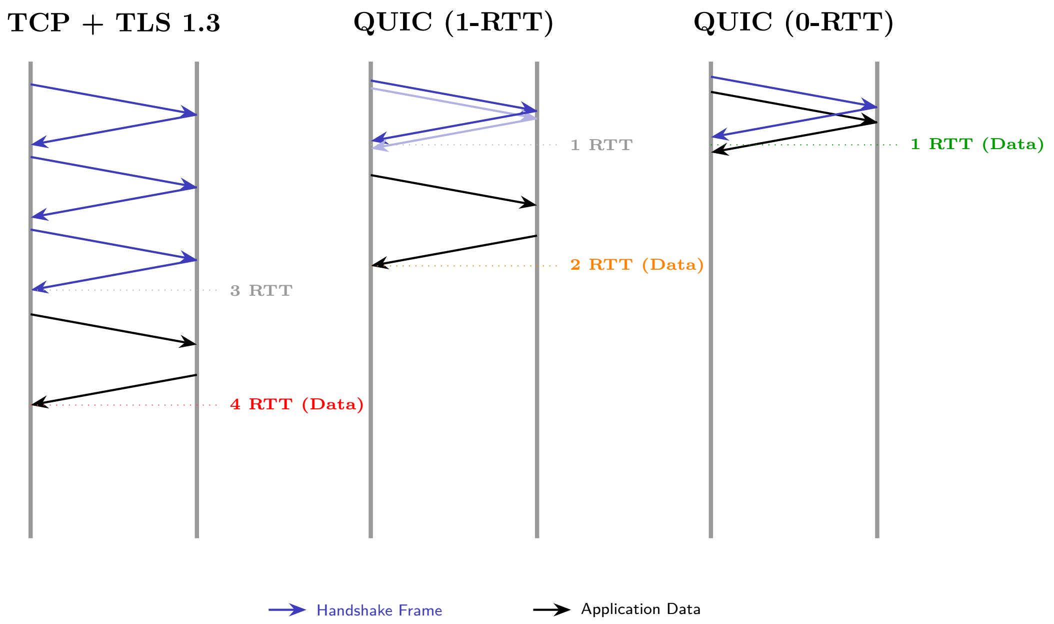


Figure 5.12: Three-way latency comparison showing the evolution of connection establishment. QUIC consolidates handshakes to save round-trips during new connections (1-RTT) and allows immediate data exchange (0-RTT) when session parameters are cached.

5.3.2.3 Connection ID and Migration

To understand connection migration, it helps to first recall how traditional TCP connections are identified. In TCP, a connection is defined by four values: the source IP address, the source port, the destination IP address, and the destination port. Together, these form what is called the *four-tuple*. As long as all four values remain unchanged, the connection is considered valid.

This design has an important consequence. If any of these values changes, the connection no longer matches the original four-tuple and is immediately terminated. In practice, this happens whenever a node's network address changes. For example, if a validator switches from a Wi-Fi network to a cellular network, its IP address changes and all existing TCP connections are dropped. The node must then establish new connections and repeat the full handshake process.

QUIC takes a different approach by separating the notion of a connection from the network path used to carry packets. Instead of identifying a connection by IP addresses and ports, QUIC introduces a *Connection ID* (CID). The CID is a value chosen by the endpoints that uniquely identifies the connection itself. Every QUIC packet carries a Connection ID. When a packet arrives, the receiver uses the CID to determine which connection state it belongs to, independently of the packet's source address or port.

If a node's network path changes—for example, due to a change in IP address—it can continue sending packets with the same Connection ID from the new address. As long as the packets are correctly authenticated, the peer accepts them as belonging to the existing connection. The connection continues without interruption, and no new handshake is required.

This ability to move an existing connection across network paths is called *connection migration*. It improves robustness in real-world networks where address changes are common, and it reduces unnecessary reconnections in long-lived peer-to-peer protocols.

5.3.2.4 Authenticated and Encrypted by Default

QUIC incorporates cryptographic protection directly into the transport protocol. All QUIC connections use encryption and authentication, and the protocol does not support unprotected operation.

This design contrasts with TCP, which provides only reliable byte delivery. In a TCP-based stack, confidentiality and authentication are provided by a separate protocol layered on top of the transport. Without such an additional layer, TCP traffic is transmitted in plaintext and can be observed or modified by intermediaries on the network path.

In QUIC, cryptographic keys are established as part of connection setup using TLS 1.3. During the initial handshake, packets are protected using temporary handshake keys derived from the TLS key exchange. These keys provide integrity and confidentiality for the handshake messages themselves.

Once the handshake completes, both endpoints derive session keys, referred to as 1-RTT keys, which are then used to protect all subsequent packets. Every packet carries an authentication tag that allows the receiver to verify its integrity and origin.

In addition to encrypting application data, QUIC encrypts most transport-level metadata. Fields such as stream identifiers and packet numbers are not visible to the network. Only minimal information required for packet routing remains unencrypted.

As a result, application data is never transmitted without integrity protection, and unauthorized modification or injection of packets is detected by the receiver. For Lean Consensus, this provides authenticated and confidential transport semantics without requiring additional security mechanisms at the application layer.

5.3.3 Connection Security and Node Identity

Once a transport connection has been established, the next step is to secure it. Securing a connection serves two fundamental purposes. First, it ensures that data exchanged between peers cannot be read or modified by third parties on the network path. Second, it allows each peer to verify the identity of the node it is communicating with.

Lean Consensus secures all connections using authenticated key exchange. When QUIC is used as the transport, this functionality is provided by TLS 1.3. When TCP is used as a fallback transport, equivalent guarantees are provided by the Noise Protocol Framework. In both cases, a confidential and authenticated channel is established before any consensus messages are exchanged.

5.3.3.1 Self-Certifying Node Identity

Lean Consensus does not rely on centralized certificate authorities, as used in the HTTPS web security model. In a peer-to-peer network, there is no global authority responsible for issuing, validating, or revoking certificates. Instead, node identity is derived directly from cryptographic keys that the node itself controls.

Each node possesses a long-term public key, which is advertised to the network as part of its Ethereum Node Record (ENR). Peers learn this public key during the discovery process, before any transport connection is established. As a result, the purpose of the connection handshake is not to discover the peer's identity, but to verify that the remote node actually controls the private key corresponding to the public key already advertised in its ENR. This approach is known as *self-certifying identity*.

5.3.3.2 Security Objectives of the Handshake

Establishing a secure session between two nodes must achieve three cryptographic objectives:

1. **Identity binding.** Each node must verify that the remote peer controls the private key corresponding to its advertised public key. This prevents impersonation attacks, in which an adversary attempts to masquerade as another node.
2. **Key agreement.** The two nodes must derive shared secret keys that will be used to encrypt and authenticate subsequent traffic. These keys must be known only to the two endpoints.
3. **Forward secrecy.** Compromise of a node's long-term identity key in the future must not allow an attacker to decrypt past communication. This requires that session keys be derived from ephemeral secrets and discarded when the connection ends.

The central cryptographic tool used to achieve key agreement and forward secrecy is the Diffie–Hellman key exchange.

5.3.3.3 Diffie–Hellman Key Agreement

The purpose of a Diffie–Hellman key exchange is to allow two nodes to derive a shared secret over a public network, even if all messages exchanged between them can be observed by an attacker.

The problem it solves is simple to state. Two nodes wish to agree on a secret value that will later be used as an encryption key. They can exchange messages freely, but they must assume that the network is completely public: every message may be copied, recorded, or replayed. The challenge is therefore to agree on a secret *without ever transmitting that secret directly*.

A shared mathematical setting. Diffie–Hellman relies on a shared mathematical structure that is fixed and public. This structure is called a *group*. At a high level, a group is a set of elements together with an operation that combines two elements to produce a third.

The operation has three essential properties:

- it is associative,
- it has an identity element,
- every element has an inverse.

For the purposes of Diffie–Hellman, the most important fact is that the operation can be applied repeatedly. Applying the operation many times corresponds to a notion of multiplication by a number. Lean Consensus uses a specific group derived from elliptic-curve cryptography, known as Curve25519. The details of elliptic curves are not required here; what matters is that Curve25519 defines:

- a finite group G,
- a well-defined operation on elements of G,
- and a publicly agreed starting element.

The generator. Within the group G, a specific public element is chosen, denoted G_0. This element is called a *generator*. Intuitively, a generator is a starting point from which many other elements of the group can be reached by repeatedly applying the group operation. Both the group G and the generator G_0 are public constants. Every participant in the network uses the same values.

Private secrets. Each node now chooses a private secret number:

$$\text{Node A chooses a secret integer } a, \qquad \text{Node B chooses a secret integer } b. \tag{5.1}$$

These numbers are chosen uniformly at random from a large range and are kept strictly private. They are never transmitted over the network.

Public key derivation. Using the public generator G_0, each node computes a public value by repeatedly applying the group operation. This repeated application is called *scalar multiplication* and is written using the symbol $\cdot$:

$$A = a \cdot G_0, \qquad B = b \cdot G_0. \tag{5.2}$$

The values A and B are the nodes' public keys for this exchange. They are sent over the network and may be observed by anyone. The critical property of the group is the following asymmetry:

- given a and G_0, it is easy to compute $A = a \cdot G_0$;
- given G_0 and A, it is computationally infeasible to recover a.

This one-way property is the foundation of Diffie–Hellman security.

Deriving the shared secret. After exchanging public keys, each node combines its own private number with the other node's public value:

$$\underbrace{a \cdot B}_{\text{computed by Node A}} = \underbrace{b \cdot A}_{\text{computed by Node B}}. \tag{5.3}$$

Expanding the definitions shows why these two values are equal:

$$a \cdot (b \cdot G_0) = b \cdot (a \cdot G_0). \tag{5.4}$$

Both nodes therefore arrive at the same group element, even though neither has learned the other's private secret. An attacker observing the network sees only the public information:

$$G_0,\ A = a \cdot G_0,\ B = b \cdot G_0, \tag{5.5}$$

and cannot compute the shared value without solving a problem that is believed to be computationally infeasible.

From group element to encryption keys. The value produced by the Diffie–Hellman exchange is a group element, not an encryption key. On its own, this value is not guaranteed to have the exact structure or format required by symmetric encryption and authentication algorithms.

To address this, the shared value is passed through a *key-derivation function* (KDF). A KDF is a deterministic algorithm that takes some input material and produces one or more cryptographic keys with well-defined length and statistical properties. Its role is to transform the raw Diffie–Hellman output into keys that are suitable for practical use. Formally, the process can be written as:

$$K = \mathrm{KDF}(a \cdot B), \tag{5.6}$$

where the input $a \cdot B$ is the shared group element derived by both nodes.

The KDF performs several important tasks at once. It extracts uniformly random-looking bits from the shared value, expands them to the required key length, and ensures that small differences in the input lead to completely unrelated outputs. As a result, the derived key K can safely be used as a symmetric encryption key or as input to message authentication algorithms.

In practice, modern protocols derive multiple independent keys from the same shared value—for example, separate keys for encryption and for integrity protection—by invoking the KDF with different context information. This separation ensures that different cryptographic functions do not interfere with each other.

5.3.3.4 Handshake Realization in TLS 1.3

The abstract mechanisms described above are realized concretely during the TLS 1.3 handshake used by QUIC. The handshake simultaneously authenticates the remote peer and performs a Diffie–Hellman key exchange to derive session keys.

Figure 5.13 shows the structure of this handshake. Each peer contributes a Diffie–Hellman key share, proves possession of its long-term identity key, and derives shared encryption keys. Application data is transmitted only after the handshake completes, and all such data is cryptographically protected.

5.4 Gossipsub: Scalable Data Propagation

With the transport layer established, nodes possess the ability to send bytes to one another securely. However, the transport layer provides only point-to-point communication. The consensus layer requires a broadcast mechanism: when a validator produces a block or an attestation, that data must reach every other interested node in the network within seconds.

A naive approach to broadcasting is flooding, where a node blindly forwards every message it receives to every peer it is connected to. While this guarantees the lowest possible latency, it is

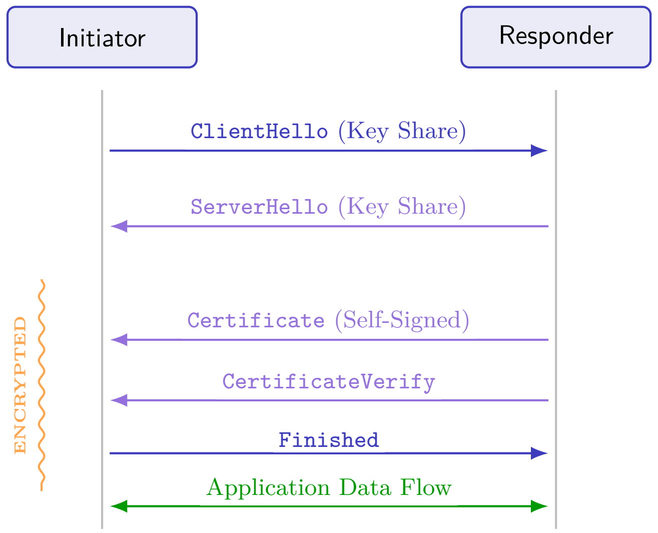


Figure 5.13: The TLS 1.3 handshake as used in Lean Consensus. The handshake authenticates the peer and derives encryption keys before any application data is transmitted.

inefficient: if every node forwards every message, the network is quickly overwhelmed by redundant traffic. The opposite approach is a structured overlay, such as a Distributed Hash Table (DHT). In a DHT, the network is organized like a structured index; to send or retrieve data, a node must route the request through a specific sequence of peers, hopping from one to the next to reach the destination. While this is highly bandwidth-efficient because messages are routed precisely rather than sprayed, each hop introduces a network round-trip delay. For a consensus protocol that must synchronize thousands of nodes within seconds, these accumulated latencies are unacceptable.

To navigate this trade-off, Ethereum utilizes *Gossipsub*, a hybrid protocol from the libp2p stack. Before examining its mechanics, we must first understand how data is organized within this system.

5.4.1 The Concept of Topics

In a pub-sub (publish-subscribe) network, data flows through logical channels called *topics*. A topic is simply a string that acts as a label for a specific stream of messages. Nodes express interest in data by subscribing to these topics.

If a node subscribes to a topic, it expects to receive all messages published to that channel. If it is not subscribed, it ignores that traffic entirely. This allows the network to be segmented: a node only pays the bandwidth cost for the data it actually needs.

In Ethereum, topics are structured strings that combine the network name, the fork version, the message type, and the encoding. For example:

```
/eth2/b5303f1a/beacon_block/ssz_snappy
```

This structure ensures strict isolation. A node on the Mainnet chain (fork digest `b5303f1a`) will never accidentally exchange blocks with a node on a testnet, because their topic strings will differ.

With this structure in place, Gossipsub achieves both speed and reliability by splitting message propagation into two distinct modes of operation, as illustrated in Figure 5.14:

1. **Eager Push (The Mesh):** For speed, every node maintains a stable set of high-bandwidth connections called a *mesh*. When a new message arrives, the node immediately pushes the full payload to these peers. This creates a fast distribution tree that functions similarly to flooding, but is contained to a small, manageable number of paths to prevent network saturation.
2. **Lazy Pull (Gossip):** For reliability, nodes also communicate with random peers outside their mesh. Instead of sending full messages, they periodically exchange metadata—lists of message identifiers they have seen. If a node discovers from this metadata that it missed a message, it explicitly requests the data. This lazy mechanism acts as a lightweight safety net, ensuring that any data dropped by the mesh is eventually recovered.

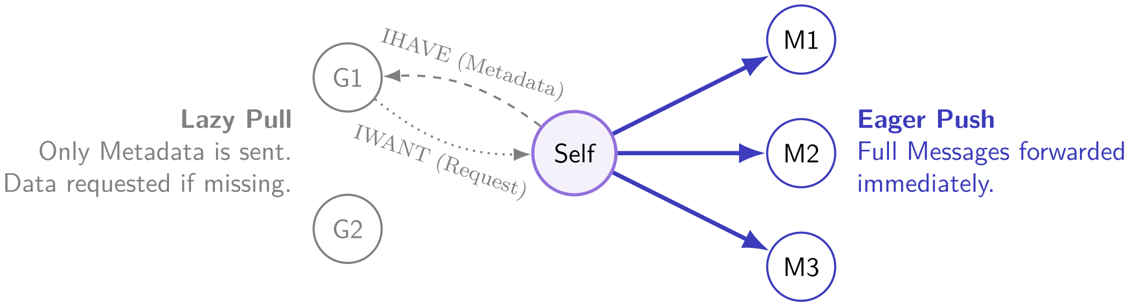


Figure 5.14: The Hybrid Architecture. The protocol uses high-bandwidth mesh links (right) for speed and low-bandwidth gossip links (left) for redundancy.

5.4.2 The Mesh and Eager Push

The primary delivery mechanism in Gossipsub is the *mesh*. For every topic, such as the beacon block topic, a node maintains a subset of its peers as mesh members. These peers form a stable, bidirectional overlay graph.

When a node receives a valid message for a topic, it immediately forwards the full message body to all peers in its mesh for that topic, excluding the sender. This is known as eager push. Because the mesh degree is bounded—typically much smaller than the total number of connected peers—the amplification factor is controlled.

The protocol attempts to maintain the size of this mesh, denoted as $|mesh|$, within a specific range defined by three parameters:

- D_{low}: The minimum number of mesh peers (default 6).
- D: The target number of mesh peers (default 8).
- D_{high}: The maximum number of mesh peers (default 12).

This design solves a critical scaling problem. By strictly limiting how many peers a node talks to (the mesh size), the protocol ensures that a node does not get overwhelmed as the network grows. Even if the total network expands from 1,000 to 100,000 nodes, a single node still only re-broadcasts messages to its handful of mesh partners, keeping its bandwidth usage constant.

At the same time, this limited connectivity does not slow down the network. Because the connections are random and dense, the path between any two nodes remains very short. A message can hop from one side of the global network to the other in just a few steps, ensuring that data spreads rapidly regardless of how large the network becomes.

5.4.3 Gossip and Lazy Pull

The mesh is highly efficient because it sends data down specific, pre-established paths. However, this structure has a weakness: if a peer in the mesh disconnects or maliciously refuses to forward a message, that specific path is broken, and everyone downstream stops receiving data. In a worst-case scenario, this could split the network into isolated islands that cannot hear each other.

To fix this, Gossipsub employs a secondary safety mechanism called *gossip*. This mechanism runs in parallel to the mesh and constantly checks for missing data by talking to random peers outside the main distribution tree.

Periodically, a node selects a random subset of peers with whom it has no mesh connection. Instead of transmitting full blocks or attestations, the node sends a lightweight control message—the *IHAVE* message—containing only the cryptographic identifiers of data it possesses. This acts as a bandwidth-efficient advertisement.

When a peer receives an *IHAVE* message, it checks the advertised identifiers against its local record of seen messages. If it identifies a gap in its history—indicating that the mesh failed to deliver a specific message—it replies with an *IWANT* request. Only then does the original sender transmit the full message body. This interaction acts as a ubiquitous safety net, ensuring that data eventually bypasses broken mesh links and reaches every participant in the network.

5.4.4 Control Messages

To manage the complex topology of the mesh and the flow of data, Gossipsub employs a specific vocabulary of control messages. These messages are rarely sent in isolation; for network efficiency, they are typically batched and piggybacked onto standard RPC frames. We can categorize these messages into three distinct operational groups: mesh management, data discovery, and traffic optimization. This is illustrated in Figure 5.15.

5.4.4.1 Mesh Management (GRAFT and PRUNE)

These two messages act as the clutch for the eager push mechanism, engaging or disengaging the high-bandwidth transmission of full messages.

- **GRAFT:** This message upgrades a connection. When a node joins a topic or its mesh size falls below the minimum threshold (D_{low}), it sends a GRAFT to specific peers. If the recipient accepts, the link is immediately upgraded to a full mesh link, and the peers begin exchanging full message bodies.

- **PRUNE:** This message downgrades a connection. If a node becomes oversubscribed (exceeding D_{high}) or leaves a topic, it must remove peers from its mesh. It sends a PRUNE message to notify them that they will no longer receive full message pushes. Crucially, a PRUNE message carries a *backoff duration*. This timer forces the recipient to wait before attempting to GRAFT back, preventing rapid oscillations where nodes repeatedly join and get kicked from the mesh.

5.4.4.2 Data Discovery (IHAVE and IWANT)

These messages power the lazy pull mechanism, allowing nodes to discuss data availability without committing bandwidth to the transfer.

- **IHAVE:** This is a metadata advertisement. A node periodically selects peers outside its mesh and sends them a list of message identifiers (hashes) it possesses. It effectively asks, “I have these items; do you need them?”
- **IWANT:** This is the explicit request. If a node receives an IHAVE advertisement containing a hash it has not seen, it responds with an IWANT message. This triggers the sender to transmit the full payload.

5.4.4.3 Traffic Optimization (IDONTWANT)

Introduced in Gossipsub v1.2, the *IDONTWANT* message is an optional optimization for large data types, such as blocks. Because the mesh is eager, a node might receive the same large block from multiple mesh peers simultaneously. This message allows a node to proactively signal to its peers that it has already received the data, preventing them from wasting bandwidth by forwarding a redundant copy.

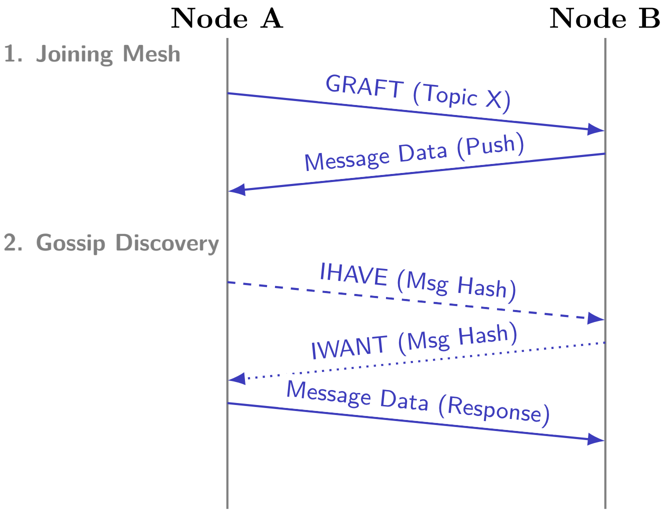


Figure 5.15: Control Message Lifecycle. **Top:** A *GRAFT* message instantly upgrades a peer, enabling eager push of data. **Bottom:** The gossip process requires a three-step handshake (*IHAVE* → *IWANT* → Data) to conserve bandwidth.

5.4.5 State Management and Caching

To implement the lazy pull mechanism and routing logic effectively, the router cannot simply be a stateless pass-through. It must maintain a memory of recent events to facilitate data retrieval and prevent loops.

5.4.5.1 The Message Cache

The lazy pull mechanism relies on a fundamental promise: if a node advertises that it has a message, it must actually be able to serve that message when requested. This requires a short-term memory storage known as the *message cache*.

Conceptually, this cache operates as a sliding window of time buckets driven by the protocol's internal clock, known as the *heartbeat*. The heartbeat is a fixed, periodic pulse (typically occurring less than once per second) that synchronizes the router's maintenance tasks. In the context of the cache, each heartbeat signifies the start of a new time interval. As these pulses occur, the windows shift: new messages enter the current newest bucket, while the history slides back, eventually causing the oldest bucket to fall off the edge and be evicted to free up memory.

This history serves two distinct purposes, dividing the cache into two logical zones:

1. **The Advertising Window:** Messages in the most recent buckets are considered fresh. The node actively includes their IDs in gossip advertisements (*IHAVE*) sent to other peers.
2. **The Retrieval Window:** This includes the fresh buckets plus a few older ones. While the node stops advertising these older messages to save bandwidth, it retains them briefly to satisfy any incoming requests (*IWANT*) from peers who might be lagging slightly behind.

Figure 5.16 illustrates this sliding window architecture.

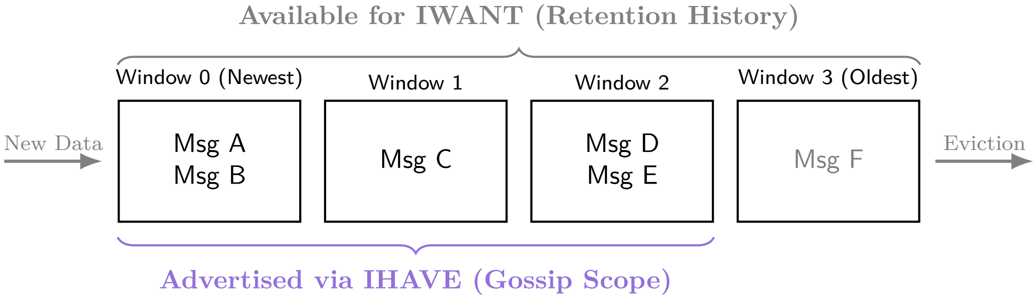


Figure 5.16: The Message Cache Sliding Window. The cache is segmented into time windows. While the node retains a longer history to answer requests (top brace), it only actively gossips about the most recent data (bottom brace) to conserve bandwidth.

5.4.5.2 The Seen Cache

In a mesh network, the same message will often arrive from multiple peers simultaneously. To prevent bandwidth waste and infinite routing loops, the router must recognize data it has already processed.

The *Seen Cache* acts as the node's long-term memory for deduplication as illustrated in Figure 5.17. It stores a set of message identifiers (hashes) accompanied by a timestamp. When a message arrives, its ID is checked against this set:

- **If found:** The message is considered a duplicate and is immediately dropped.
- **If new:** The message is marked as seen, processed, and forwarded to mesh peers.

Entries in this cache have a Time-to-Live (TTL) significantly longer than the message cache. This ensures that a node does not accidentally re-process or re-broadcast an old block, even if it circles back through the network minutes later.

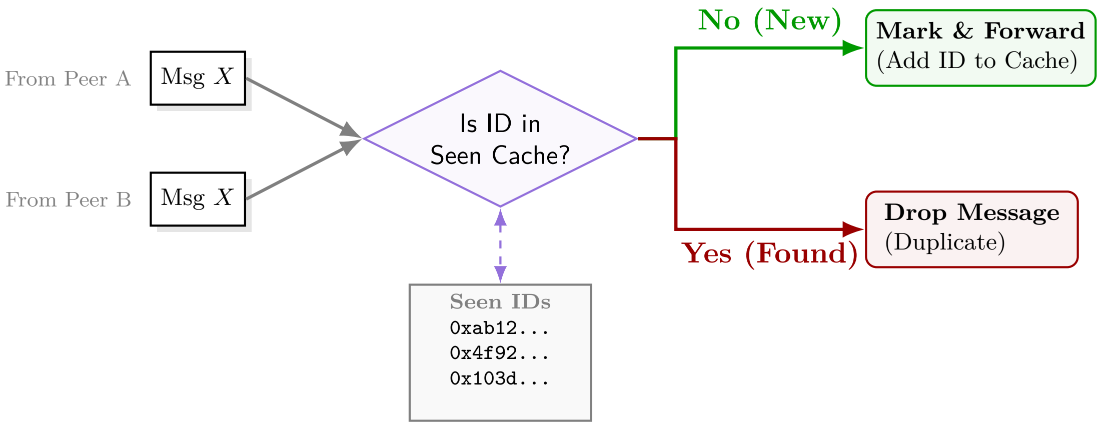


Figure 5.17: The Deduplication Logic. The router acts as a filter. The first copy of a message (e.g., from Peer A) passes through, while subsequent copies (e.g., from Peer B) match the cache and are dropped.

5.4.5.3 Fanout State

A fundamental design principle of Gossipsub is that a node normally only pushes data to peers in its mesh. However, this creates a dilemma: how does a node publish a message to a topic it does not want to listen to?

In Ethereum, this is a frequent requirement. A validator often needs to broadcast a specific vote (attestation) on a specific subnet. However, because the validator is not assigned to aggregate votes on that subnet, it has no interest in paying the bandwidth cost to subscribe to it. It wants to speak without having to listen. Since the node has no mesh for that topic, it cannot use the standard eager push mechanism. Instead, it utilizes a temporary mechanism called the *Fanout*.

When a node attempts to publish to a topic it is not subscribed to, it performs the following logic:

- **Selection:** The node scans its connection list for peers that are subscribed to the target topic. From this pool, it randomly selects a subset—equal in size to the standard mesh degree (D)—to act as a temporary bridge into the network.
- **Propagation:** It pushes the message to these peers. Because these peers *are* subscribed to the topic, they will immediately forward the message into the actual mesh, acting as the entry point for the data.
- **Retention:** The node maps these peers to the topic in its local *Fanout State*. If the node publishes another message to the same topic shortly after, it reuses the same peers to maintain a stable topology.

This state is ephemeral. If the node stops publishing to that topic for a set duration (a timeout), the fanout state is discarded to save memory. Conversely, if the node decides to fully subscribe to the topic, the peers in the fanout state are effectively promoted to become the initial members of the new mesh.

5.4.6 The Heartbeat Procedure

A pure event-driven system—one that only reacts when a message arrives—can be unstable. Peers might disconnect silently, leaving a node isolated without it realizing. Conversely, a node might accumulate too many connections during a burst of activity, draining its bandwidth.

To maintain stability, Gossipsub relies on a periodic maintenance cycle known as the *heartbeat*. Think of this as the protocol's internal clock. In Lean Consensus, this clock ticks approximately every 0.7 seconds (700 milliseconds).

As illustrated in Figure 5.18, during each tick, the router pauses to evaluate the health of its connections and perform three essential maintenance tasks for every topic it is subscribed to:

1. **Mesh Maintenance (Topology Repair):** First, the node checks if its mesh has drifted away from the ideal size.

   - **Recruitment (Too Few Peers):** If the number of mesh peers has dropped below the minimum threshold ($|mesh| < D_{low}$), the node is at risk of isolation. It selects candidate peers from the general pool and sends them *GRAFT* messages to recruit them into the mesh, bringing the count back up to the target.
   - **Trimming (Too Many Peers):** If the mesh has grown too large ($|mesh| > D_{high}$), perhaps due to many incoming graft requests, the node is wasting bandwidth. It identifies the excess peers—typically selecting those with the lowest behavioral scores—and sends them *PRUNE* messages to disconnect them from the mesh.

2. **Gossip Emission (The Safety Net):** Next, the node performs the lazy pull mechanism described earlier. It selects a random set of peers (D_{lazy}) that are *not* part of its mesh. It then sends them an *IHAVE* message containing IDs from its message cache. This keeps the wider network informed about the data available, ensuring that even peers without a direct mesh connection can eventually retrieve missing blocks.

3. **Housekeeping (Memory Management):** Finally, the node advances time. It shifts the sliding windows of the message cache (evicting the oldest history), cleans up expired entries in the seen cache to free up memory, and discards any Fanout states that have timed out.

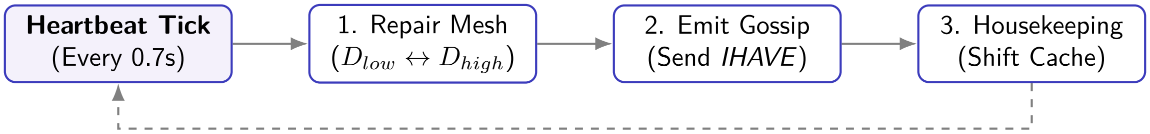


Figure 5.18: The Heartbeat Cycle. Mesh topology and lazy gossip are not real-time event responses but are regulated by this periodic clock, preventing network thrashing.

5.4.7 Ethereum-Specific Message Identification

For the gossip protocol to effectively filter duplicates, every message must have a globally unique identifier (Message ID). In a standard implementation, this ID is often just a hash of the raw bytes received over the wire. However, Ethereum introduces a complexity that makes this naive approach insecure: *compression*.

Ethereum uses the Snappy algorithm to compress data before transmission. This creates a potential vulnerability known as *aliasing*. Because compression algorithms can produce slightly different binary outputs for the same input (depending on compression levels), a malicious actor could validly compress the same block in ten different ways. If the ID were based on the compressed bytes, the network would treat these as ten distinct messages, bypassing the deduplication cache and flooding the network.

To prevent this cache pollution, Ethereum defines a custom Message ID algorithm that enforces a canonical identity for every message, regardless of its representation on the wire.

5.4.7.1 The Identity Function

The protocol calculates the ID by mixing metadata with the payload content. The formula is defined as:

$$\text{Message ID} = \text{SHA256}(\text{domain} \parallel \text{len}(\text{topic}) \parallel \text{topic} \parallel \text{data})[:20]$$

This function produces a 20-byte identifier. The critical components here are the `data` and the `domain`, which interact to ensure security.

5.4.7.2 The Domain Separator

The `domain` is a 1-byte prefix that acts as a switch, determining how the payload should be interpreted. Its value depends entirely on whether the message can be successfully decompressed.

- **Valid Compression (Domain `0x01`):** If the incoming message successfully decompresses using Snappy, the node sets the domain to this valid tag. Crucially, it then hashes the uncompressed data. This ensures that even if two peers send the same block with slightly different compression, they both map to the same ID, allowing the router to recognize them as duplicates.
- **Invalid/No Compression (Domain `0x00`):** If Snappy decompression fails (or if the node chooses not to decompress), the domain is set to this invalid tag, and the hash is calculated over the raw wire bytes. This ensures that malformed or attack packets still have a deterministic ID, allowing them to be tracked and blacklisted without crashing the decompression engine.

5.5 The Request-Response Domain

Gossipsub provides a mechanism for the rapid dissemination of *new* data. However, a consensus protocol also requires a mechanism to retrieve *historical* or specific data. When a node first joins the network, it must download the entire history of the chain to synchronize with the head. Similarly, if a node misses a specific block during gossip due to network packet loss, it requires a surgical method to request exactly that missing piece.

To satisfy these requirements, Ethereum implements a direct Request-Response (Req/Resp) domain. Unlike the multicast nature of gossip, this domain establishes a direct, 1-to-1 conversation between two peers. It follows a client-server model where one peer issues a specific query (e.g., "Send me blocks 100 to 200") and the other responds with the requested data or an error code.

5.5.1 Protocol Identification and Negotiation

Because the peer-to-peer connection is multiplexed, it functions as a shared conduit for various distinct conversations. At any given moment, a node might need to simultaneously download historical blocks to synchronize its chain, exchange status messages to verify a peer's fork compatibility, or issue liveness checks to ensure the connection remains active.

To disambiguate these interleaved data streams, the network assigns a unique Protocol ID to each specific message type. Unlike traditional numeric port numbers, this identifier is a structured text path that serves as a precise label for the data stream, defining the protocol family, the specific operation, the schema version, and the encoding format.

For example, the identifier for requesting a range of blocks is constructed as follows:

```
/eth2/beacon_chain/req/beacon_blocks_by_range/1/ssz_snappy
```

This string provides the receiver with a complete instruction set for parsing the incoming bytes, where each segment acts as a specific configuration parameter:

- **Interaction Domain (`/req`):** This segment classifies the traffic as a Request-Response interaction. It distinguishes direct queries from other traffic types like gossip or discovery, ensuring the router dispatches the stream to the appropriate subsystem.
- **Message Schema (`/beacon_blocks_by_range`):** This specifies the exact data structure being exchanged. It tells the application layer to prepare a handler capable of fetching a sequence of blocks from the database and serializing them into the expected response format.
- **Schema Version (`/1`):** This ordinal integer dictates the version of the data structure. Unlike semantic versioning, this is a strict compatibility check; a node supporting only version 2 must reject a version 1 request to prevent deserialization errors or logic mismatches.
- **Encoding Strategy (`/ssz_snappy`):** This final segment mandates the wire format. It acts as a contract enforcing that the payload must first be serialized using Simple Serialize (SSZ) and then compressed using Snappy framing, allowing the receiver to configure its decoding pipeline correctly before reading a single byte.

When a node wishes to initiate a request, it cannot simply begin transmitting data. It must first utilize a sub-protocol called `multistream-select` to establish a common language. The initiator proposes a specific Protocol ID—effectively asking, "I would like to speak `BlocksByRange` Version 1 using Snappy compression." The responder accepts this proposal only if it supports that exact version and encoding. This mandatory negotiation step acts as a compatibility guardrail, ensuring that two nodes never attempt to exchange data unless they have explicitly agreed on the format of the conversation.

5.5.2 The Stream Lifecycle

The Req/Resp domain operates on a stream-per-request model. This means that streams are single-use tools: they are created for a specific query and discarded the moment that query is resolved.

This design differs significantly from protocols like HTTP/1.1, which often keep a connection open to send multiple requests sequentially. By isolating every request into its own ephemeral stream, Ethereum ensures that a heavy background task (like downloading 50 historical blocks) never blocks a high-priority, latency-sensitive message (like a status check).

As illustrated in Figure 5.19, the life of a stream follows a strict four-step narrative defined by an asymmetric closure pattern:

1. **Initiation:** The Requester begins by negotiating the protocol and opening a new stream. It does not wait for an acknowledgement; to minimize latency, it immediately writes the full, encoded request payload into the stream.
2. **Half-Closure:** This is the most critical distinct feature of the protocol. The moment the Requester finishes writing the payload, it closes the *write side* of the stream.

   This does not sever the connection. Instead, it sends a formal *End-of-File (EOF)* signal to the Responder. This acts as a cryptographic "Over": it explicitly tells the Responder, "I have finished speaking, and I am now waiting for your reply."
3. **Processing and Response:** The Responder reads the incoming stream until it encounters the EOF signal. This confirms that the full request has been received. The Responder then processes the query (e.g., fetching blocks from its database) and writes the result back to the stream. For requests that return multiple items, the data is sent as a sequence of discrete chunks.
4. **Termination:** Once the Responder has transmitted the final chunk of data, it closes its write side. Because the Requester had already closed its side in Step 2, the stream effectively transitions to a *Fully Closed* state and is discarded by both nodes.

5.5.3 The Wire Format

To ensure bandwidth efficiency and protect nodes from Denial-of-Service (DoS) attacks, the protocol enforces a strict serialization schema known as `ssz_snappy`. This format acts as a transport envelope, wrapping the logical consensus objects (such as blocks or attestations) in a protective layer that handles both compression and safety delimiting.

As visualized in Figure 5.20, the wire format consists of three distinct layers, processed sequentially by the receiver:

1. **Response Code (1 Byte):** Every response chunk begins with a single status byte. This acts as a header signal, allowing the responder to indicate the success or failure of a specific chunk without disrupting the parsing logic of the underlying stream.

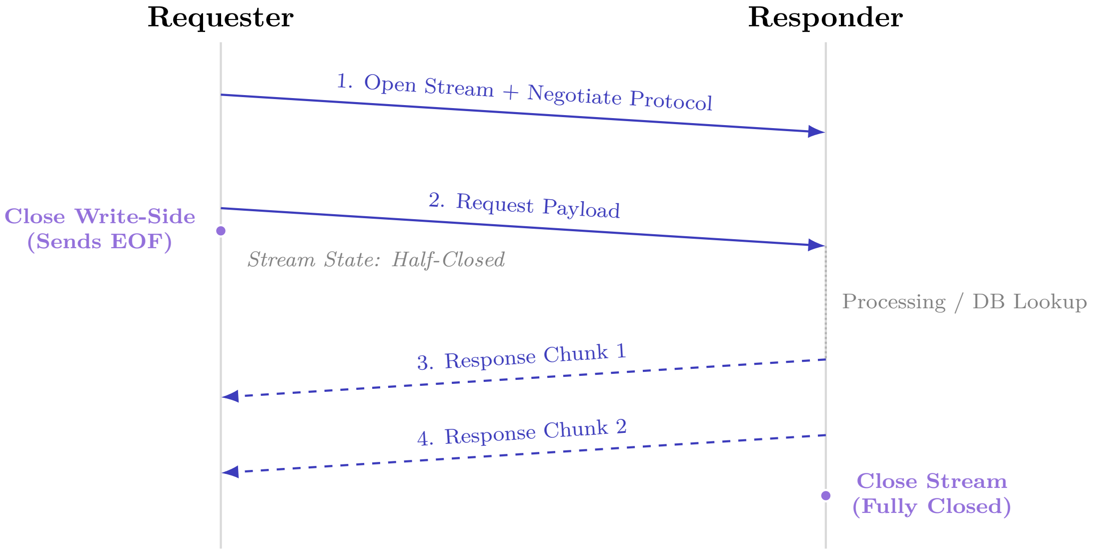


Figure 5.19: The Req/Resp Stream Lifecycle. The diagram highlights the asymmetric closure. The Requester (left) initiates the shutdown by closing its write-side immediately after sending the request (Step 2). This signals completion to the Responder, who processes the data and closes the stream only after sending all chunks (Step 4).

- `0x00` (Success): The chunk contains valid payload data.
- `0x01` (Invalid Request): The request was malformed or violated protocol rules.
- `0x02` (Server Error): The responder encountered an internal processing fault.
- `0x03` (Resource Unavailable): The specific piece of data requested (e.g., a block root) could not be found.

2. **Varint Length Prefix:** Immediately following the status byte is the length of the *uncompressed* payload, encoded as an unsigned LEB128 Varint (Variable Integer). This prefix is a critical security feature. It forces the sender to declare the exact memory requirements of the object *before* the receiver attempts to decompress it.

   If this declared length exceeds the protocol's safety limit (typically 10 MiB), the receiver can immediately terminate the stream. This preemptive check neutralizes "decompression bomb" attacks, where a small malicious packet expands into gigabytes of data to crash the receiver's memory.

3. **Snappy Framing:** The payload itself is serialized using Simple Serialize (SSZ) and then compressed using Snappy framing. Snappy is selected over higher-compression algorithms (like Gzip) because it prioritizes decompression speed and low CPU overhead—essential characteristics for a node that must synchronize gigabytes of blockchain history in real-time.

5.5.4 Key Application Protocols

With the transport mechanics established by the wire codec (which handles compression and framing), the node requires specific logic to interpret the data. The Application Layer defines these

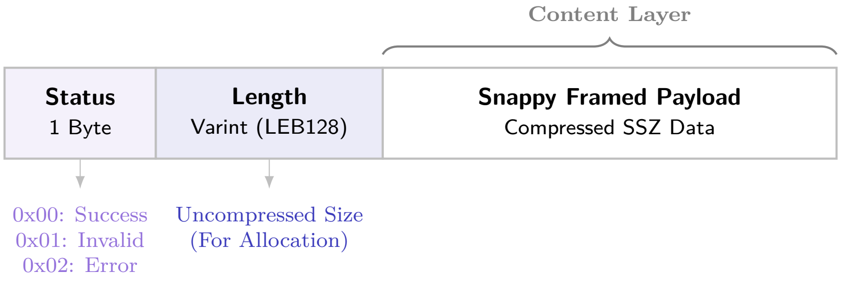


Figure 5.20: The Req/Resp Wire Format. The format is designed for stream processing. The Varint length prefix allows the receiver to allocate memory safely before expending CPU cycles on Snappy decompression.

distinct sub-protocols, each tailored for a specific consensus task. Among the various request types, two form the fundamental mechanism for joining and tracking the network.

5.5.4.1 Status: Verifying Compatibility

The moment a transport connection is established, the node faces an immediate decision: is this new peer useful? There is no value in maintaining a connection with a node that is on a different network (e.g., a testnet) or one that is hopelessly stuck on an old chain.

To resolve this, the protocol enforces a mandatory handshake called the `Status` exchange. Before any blocks or transactions are transmitted, both peers must swap a status report containing three critical identifiers. As illustrated in Figure 5.21, this data drives two distinct logic gates:

- **Fork Digest:** A 4-byte fingerprint derived from the genesis block and the current fork version. This acts as a strict firewall. If these bytes do not match exactly, the peers are effectively on different universes (e.g., Mainnet vs. Testnet) and the connection is severed immediately.
- **Finalized Checkpoint:** The latest slot the node considers immutable. This allows the peers to compare the security depth of their chains. If a peer's finalized checkpoint is incompatible with the local node's history, it indicates a deep chain split.
- **Head Slot:** The absolute tip of the node's chain. This serves as a liveness indicator. Once the network compatibility is verified via the Fork Digest, the Head Slot determines the direction of data flow.

If Peer A reports a higher head slot than Peer B, Peer B instantly recognizes that it is lagging. It effectively says, "You are ahead of me; I will start downloading blocks from you to catch up."

5.5.4.2 `BlocksByRange`: Synchronizing History

When a node identifies a gap in its history, it uses `BlocksByRange` to retrieve the missing data. This is the primary protocol for bulk synchronization. The request structure is precise: the node specifies a start slot and a count (e.g., "Send me 50 blocks starting from slot 1,000"). The peer responds with a list of signed blocks.

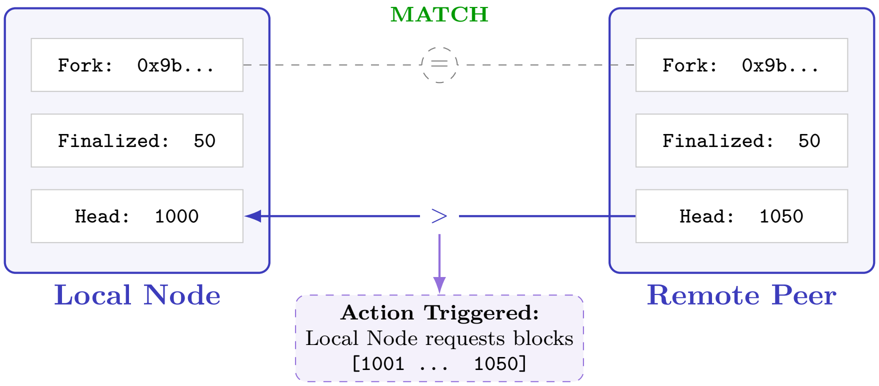


Figure 5.21: The Status Handshake Logic. The exchange serves two purposes sequentially. First, the Fork Digest acts as a gatekeeper; if it matches, the connection is accepted. Second, the Head Slot is compared to determine which peer holds new data, triggering the synchronization process.

A critical detail in this process is the handling of *empty slots*. In Ethereum, the blockchain is not a perfectly contiguous array; if a validator goes offline and misses their turn, no block is produced for that slot. The protocol explicitly allows the responder to omit these empty slots from the response to save bandwidth. Consequently, the requester cannot simply assume the received blocks are sequential. It must actively reconstruct the chain by verifying the cryptographic lineage—ensuring that the parent root of each incoming block matches the hash of the previous one—to confirm that the history is valid and unbroken.

Chapter 6

Consensus Mechanisms

The state transition function Υ defines what it means for a block to be *valid*: given a state S_n and an incoming block B, every honest node can deterministically compute the next state S_{n+1}. Validity alone, however, does not produce a blockchain. In a live network, many valid blocks may be proposed for the same slot, different nodes may learn about them in different orders, and temporary forks are unavoidable under realistic network delays. A consensus mechanism is the layer of rules and messages that allows the network to converge on a *single* history among all valid candidates.

Ethereum's Proof-of-Stake consensus achieves this convergence by having validators repeatedly publish cryptographic votes about the chain. These votes are not informal opinions. They are structured, signed statements that (i) identify which block should be considered the current head of the chain and (ii) indicate how finality should advance through checkpoints. When enough votes align, the protocol can both resolve short-term ambiguity (which fork to build on next) and lock in long-term history (which checkpoints are irreversible).

This part focuses on the mechanics of that agreement process. We treat block proposals as the introduction of candidate blocks, and validator voting as the mechanism that selects among them. We explain what validators are signing, how votes propagate through the network, how many votes are compressed into aggregates, and how these aggregates are interpreted by the protocol rules.

6.1 The Layered Structure: Execution vs. Consensus

Voting only makes sense once there is something concrete to vote *about*. In Ethereum's post-merge architecture, a block is a composite object: it contains an execution payload (transactions and the resulting application-state transition) and it is embedded in a consensus structure that defines how blocks are ordered, how forks are resolved, and how finality advances. Attestations sit exactly at this boundary. They turn a validator's local verification of a proposed state transition into a network-wide signal about which valid block should become part of the canonical history.

This division is reflected in the separation between the execution layer and the consensus layer, shown in Figure 6.1. The execution layer defines the deterministic computation that updates user-facing state: account balances, smart contract storage, and transaction processing under the Ethereum Virtual Machine. Transactions submitted by users are collected, ordered, and assembled into an execution payload. When a validator receives a proposed block, it re-executes the payload locally and verifies that all rules are satisfied (signatures are valid, balances are sufficient, and state changes follow the EVM specification). If execution does not match the claimed transition, the block is invalid and cannot be adopted.

The consensus layer is responsible for selecting a single chain among all valid proposals. It does not interpret transaction semantics; instead, it specifies how validators coordinate to choose the head of the chain and how checkpoints become justified and finalized. A block becomes canonical only when sufficient attestations support it under these rules.

When a validator signs an attestation, the signature therefore expresses two commitments at once. First, it confirms that the execution payload is valid under the execution rules. Second, it declares that, according to the validator's local fork-choice view, this block is the correct head of the chain at that slot. In this way, many independent local executions are converted into a coherent global decision about ordering and finality.

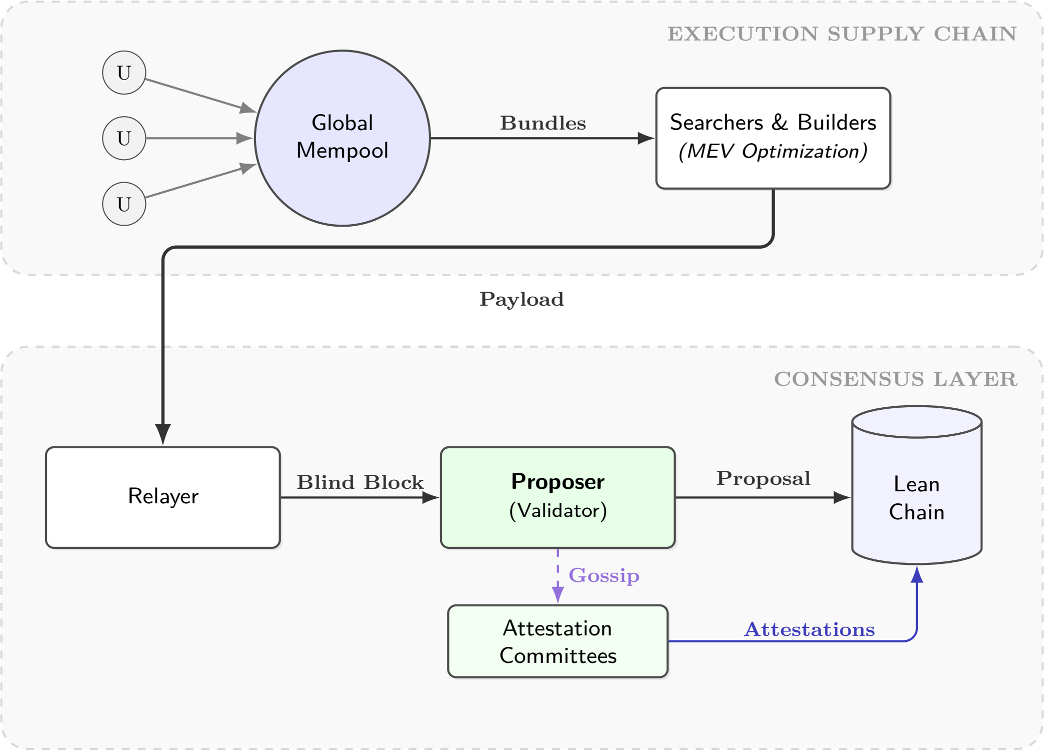


Figure 6.1: The Transaction Lifecycle. The diagram illustrates the separation between the Execution supply chain (top)—where transactions are aggregated and optimized by builders—and the Consensus Layer (bottom), where validators propose and attest to these blocks to achieve finality.

6.2 Attestation Data and the Source–Target–Head Relationship

Validators express their votes by signing a structured object called `AttestationData`. This container captures the validator's view of the blockchain at a specific slot and uniquely determines how the vote will be interpreted by the consensus protocol. Because consensus decisions depend on precise relationships between blocks and checkpoints, attestations must follow a fixed schema.

An attestation specifies both a moment in protocol time and three reference points in the chain that together describe the validator's position on past agreement, pending finality, and the current chain tip. The `AttestationData` structure therefore contains the following fields:

- **Slot.** The slot for which the attestation is produced. Since protocol time advances in discrete slots, this field ensures that the vote is evaluated in the correct temporal context and cannot be replayed in other slots.
- **Source Checkpoint.** The most recent checkpoint that the validator considers justified. This reference anchors the vote in a portion of the chain that has already achieved consensus. By building on an already justified checkpoint, the validator signals that its vote extends the established canonical history rather than endorsing an alternative fork.
- **Target Checkpoint.** The checkpoint at the boundary of the current epoch that the validator believes should be justified next. Votes for the same source–target pair accumulate vote count under the finality mechanism. When a supermajority of validators supports the pair, the target checkpoint becomes justified and may later be finalized.
- **Head Block (or Head Checkpoint).** The block that the validator considers the current head of the canonical chain. This expresses the validator's fork-choice vote and determines which branch should be extended when competing forks exist.

Together, the source, target, and head references form a single coherent statement linking past agreement, proposed finality, and the present chain tip. As illustrated in Figure 6.2, an attestation simultaneously advances the chain toward finality by voting from the source to the target, while also selecting the head block that should guide immediate chain growth.

To preserve consistency, the protocol enforces an ordering constraint among these references: the target must not precede the source, and the head must not precede the target. This requirement ensures that attestations reinforce a single chain of history. Finality progresses checkpoint by checkpoint through source–target votes, while head votes determine the block-by-block evolution of the chain between checkpoints.

An individual validator's attestation consists of this shared data together with the validator's identity and signature. When multiple validators produce identical `AttestationData`, their signatures can be aggregated into a single message accompanied by a bitfield indicating participation. This aggregation mechanism allows the protocol to collect large numbers of votes efficiently, enabling consensus to scale to a large validator set without proportional increases in bandwidth or storage.

6.3 Attestation Lifecycle

An attestation is not a static object; it is a short-lived message that is born, propagated, compressed, and finally consumed by the state transition function. Within each slot, proposers introduce new candidate blocks, while committees of validators react by producing votes about what they have observed. Those votes must move quickly: they are most valuable when they arrive early enough to influence the next proposer and to advance checkpoint justification without delay.

Although many implementation details of networking are abstracted behind the gossip layer, the lifecycle of an attestation closely follows the internal timing of a slot. Each slot is divided into five short intervals that structure how information flows through the system, culminating in inclusion during the following slot.

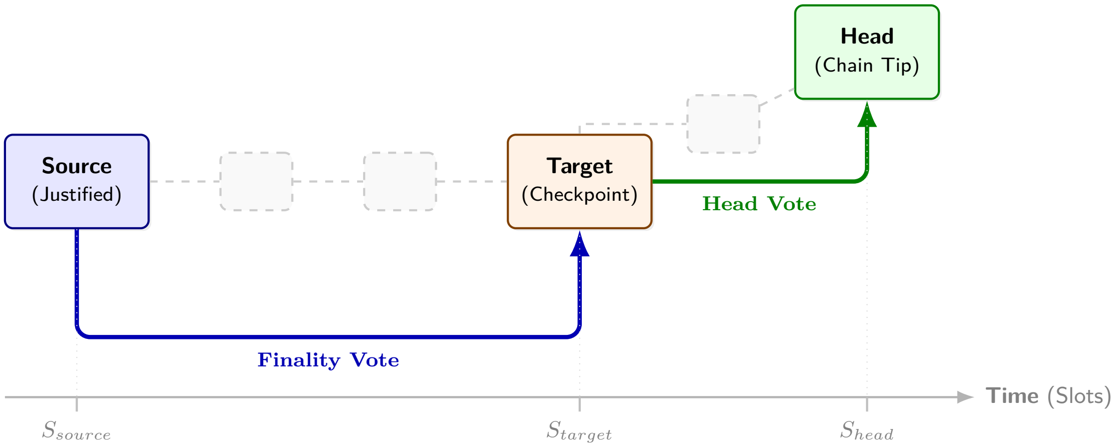


Figure 6.2: The validator casts a complex vote that links the anchored past (Source) to a proposed checkpoint (Target) for finality, while simultaneously selecting a specific leaf block (Head) for the immediate fork choice rule.

1. **Block proposal (Interval 0).** At the start of slot N, the designated proposer broadcasts a new block to the network. This block becomes the reference point against which the slot's committee will vote. If no block arrives in time, validators fall back to the best available head from their local view.

2. **Attestation creation (Interval 1).** After observing the proposal (or selecting a fallback head), each validator determines the chain tip it would extend and the appropriate source–target checkpoints for finality. From this local view it constructs the corresponding `AttestationData` and signs it, thereby committing to both a fork-choice decision (the head) and a finality vote (the source–target link). These individual attestations are disseminated within committee-specific gossip channels so that relevant peers can collect them efficiently.

3. **Aggregation (Interval 2).** Within each committee, some validators are designated (per slot) as aggregators. Their job is to collect attestations that carry *the same* `AttestationData` and merge them into a single aggregate. Aggregation happens at the message layer: the underlying vote is unchanged, but many individual signatures are combined into one, and participation is recorded compactly. The resulting aggregates are then broadcast to the wider network through global gossip, so that all peers—including the proposer of the next slot—can observe them. This stage is what makes thousands of validators voting every slot feasible in practice.

4. **Safe target computation (Interval 3).** Each node computes the deepest block that has received support from a supermajority of validators. This block becomes the safe target: the checkpoint that validators will reference as their attestation target in subsequent slots. Only blocks meeting the two-thirds threshold qualify, ensuring that the safe target reflects genuine network-wide agreement.

5. **Attestation acceptance (Interval 4).** Newly received aggregated attestations are migrated into the node's active attestation pool and the fork-choice head is updated accordingly. This migration step separates freshly arrived gossip from data that has been validated and committed to the local view, preventing in-flight messages from prematurely influencing head selection.

6. **Inclusion and processing (next slot).** Aggregates are most valuable when they are available to the next proposer. A proposer assembling block $N+1$ selects aggregates from its active

pool and includes them in the block body. Once included, they are no longer ephemeral gossip messages: they become protocol inputs to Υ. During block processing, each included aggregate is verified and then applied to the consensus accounting, increasing vote count for the relevant checkpoints and contributing to fork choice.

Figure 6.3 summarizes this progression along the slot timeline.

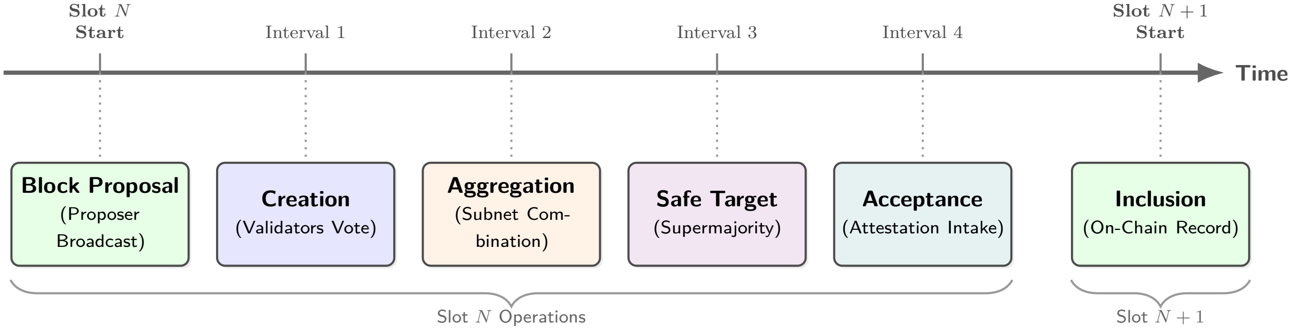

Slot N Start
Interval 1
Interval 2
Interval 3
Interval 4
Slot $N+1$ Start
Time
Block Proposal (Proposer Broadcast)
Creation (Validators Vote)
Aggregation (Subnet Combination)
Safe Target (Supermajority)
Acceptance (Attestation Intake)
Inclusion (On-Chain Record)
Slot N Operations
Slot $N+1$

Figure 6.3: The Lifecycle of an Attestation. A block proposal in Interval 0 triggers committee votes in Interval 1. These votes are aggregated within subnets during Interval 2. A safe target is computed during Interval 3, and attestations are accepted in Interval 4. Aggregated attestations are then included by the proposer of the subsequent Slot $N+1$.

6.4 Aggregation: Scaling Validator Voting

The attestation lifecycle assumes that votes can circulate efficiently across the network. In practice, the number of validators makes naïve broadcasting impossible. If every validator sent an individual signature every slot to all peers, network traffic would grow linearly with the validator set and quickly overwhelm bandwidth. Consensus based on frequent voting therefore requires a way to compress many individual votes into a single compact signal.

Ethereum achieves this by structuring attestations so that many validators often sign exactly the same message. When validators observe the same head block and the same source and target checkpoints, their votes are identical in content even though they come from different participants. This redundancy allows the protocol to merge many attestations into one object without losing information about the underlying agreement.

Aggregation works in two steps: grouping identical votes and compressing their signatures.

6.4.1 Aggregation by Data

Validators in the same committee typically produce identical attestations during a slot. Aggregators collect these matching votes and replace them with a single message containing the shared attestation data, along with compact information about which validators participated. Instead of relaying many duplicate messages, the network carries one representative vote.

6.4.2 Signature Aggregation

Because each attestation includes a signature—and XMSS signatures are relatively large—these signatures must also be compressed. Aggregators combine signatures on the same message into a single proof that attests to collective agreement. This proof can be verified against the public keys of all participants and the shared data, so one object now represents many votes.

6.4.3 Recording Participation

Even after compression, the protocol must still track who voted. This is done using a compact bitfield indexed by validator. A bit set to 1 indicates participation; 0 indicates absence. From this bitfield, a verifier reconstructs the participant set and checks the aggregated proof accordingly.

6.4.4 From Individual Votes to On-Chain Aggregates

Aggregation fits naturally into the attestation lifecycle: validators produce individual votes, aggregators merge identical ones, and the resulting aggregates are broadcast so that the next proposer can include them in a block. By the time an aggregate appears on chain, it represents the combined opinion of many validators rather than a single voice.

Figure 6.4 illustrates this transformation. Multiple validators sign the same attestation data, their signatures are merged into one proof, and the aggregate becomes the object recorded in the block. When processed, its combined voting power contributes to fork choice and checkpoint justification.

Aggregation is therefore what makes large-scale validator participation feasible without overwhelming the network.

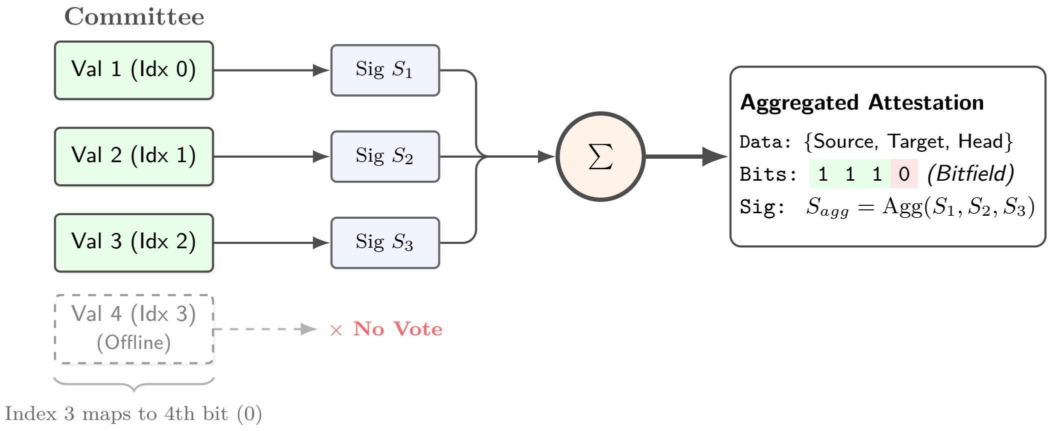


Figure 6.4: Aggregation compresses many identical attestations into one. The aggregate carries the shared `AttestationData`, a bitfield indicating which validators participated (1 for present, 0 for missing), and a single aggregated signature proof.

6.5 Validity Conditions

An attestation contributes to consensus only if every honest node can interpret it in exactly the same way. A correct signature alone is not sufficient. The vote must refer to blocks that actually exist, respect the protocol's notion of time, and align with the node's current view of justified and finalized checkpoints. Without these safeguards, malicious actors could inject votes for nonexistent forks, exploit timing edge cases, or flood the network with messages that will never influence consensus.

To prevent this, attestations are checked at several stages as they move through the system. Early checks protect the peer-to-peer layer from relaying useless data, while later checks ensure that only meaningful votes affect fork choice and finality. Conceptually, these validations fall into four layers:

- **Gossip validation** determines whether a received attestation is worth accepting and relaying at all. The node verifies that the message is well-formed, refers to known chain data, and is plausibly timed.
- **Block-level verification** determines whether a block's included attestations are authentic. Before executing the state transition, the node checks that each attestation is accompanied by a valid cryptographic proof for the claimed participants.
- **State-transition processing** determines whether the vote is still relevant in the state being applied. Even valid attestations may be stale or redundant; these checks decide whether the vote should influence fork choice and finality or be ignored.
- **Block-building filters** determine which attestations a proposer should include when constructing a new block, favoring those that other nodes can verify and that will meaningfully contribute to consensus.

6.5.1 Gossip Validation

The gossip network is the entry point for attestations and the place where invalid messages are cheapest to discard. When a node receives an attestation—whether an individual vote or an aggregate—it performs a small set of admission checks before storing or relaying it. If the message fails any of these checks, it is simply dropped.

6.5.1.1 Anchored in known chain data

A node can only relay votes it understands. The attestation must therefore reference blocks that already exist in the node's local view of the chain. In particular, the head block and the source and target checkpoints must correspond to known block roots. Without these references, the node cannot verify ancestry relationships, cannot retrieve the validator registry needed for signature verification, and cannot incorporate the head vote into fork choice.

This requirement also prevents phantom voting, where attackers attempt to inject votes for nonexistent branches in order to create confusion or waste bandwidth.

6.5.1.2 Consistent with protocol time

Attestations are statements about a specific moment in the protocol timeline. A vote for a far-future slot or one that links checkpoints out of temporal order cannot be interpreted meaningfully.

Two constraints enforce time consistency. First, the attestation's slot must be close to the node's current slot clock, typically allowing at most a one-slot lead to tolerate minor clock differences:

$$\texttt{data.slot} \leq \texttt{current slot} + 1. \tag{6.1}$$

Second, the referenced checkpoints must follow the forward flow of time. The source anchors the past, the target advances finality, and the head reflects the current chain tip, so their slots must satisfy

$$\texttt{source.slot} \leq \texttt{target.slot} \leq \texttt{head.slot}. \tag{6.2}$$

Together, these conditions ensure that the vote describes a coherent progression from established history toward the present chain. This temporal relationship is illustrated in Figure 6.5.

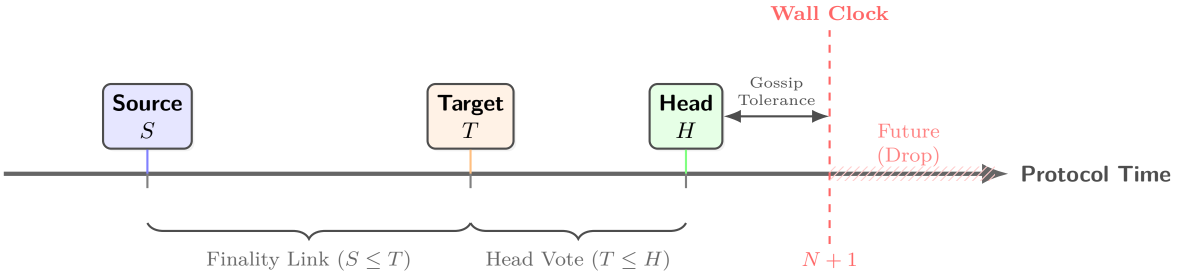


Figure 6.5: The Geometry of a Valid Vote. To be consistent with protocol time, an attestation must respect ordering: the source must precede or equal the target, and the target must precede or equal the head. Furthermore, the vote cannot refer to a slot significantly ahead of the node's local clock.

6.5.1.3 Authenticated by the claimed signer(s)

The final admission check at the gossip layer ensures that the vote was actually produced by the validators it claims to represent. A node never trusts the identity of the sender; it trusts only cryptographic evidence that binds specific validator keys to specific attestation data at a specific slot.

For an individual attestation, the node uses the validator index included in the message to locate the corresponding public key in the validator registry of the relevant state. It then verifies the signature over the attestation data. This confirms that the named validator endorsed exactly this statement about the head, source, and target at that moment in time.

For an aggregated attestation, the same principle applies to a group rather than a single signer. The participation bitfield specifies which validators are claimed to have contributed. The node expands this bitfield into a concrete set of validator indices, retrieves their public keys, and verifies the single aggregated proof against the shared attestation data. If the participant set is empty, contains invalid indices, or the proof does not validate for precisely that set, the message is rejected as malformed.

6.5.2 Block-Level Verification

The role of an attestation changes once it is embedded in a block. At the gossip layer, a faulty message can be ignored locally; inside a block, the same fault would compromise a candidate for the canonical chain. For this reason, nodes enforce a set of strict invariants before attempting the state transition. Any failure at this stage causes the entire block to be rejected.

The first requirement is structural consistency. A block contains both attestation data and the cryptographic proofs that authenticate them. These collections must correspond exactly, ensuring that every attestation is accompanied by the proof needed to verify it.

The second requirement is proof validity. Even if signatures were checked earlier during gossip propagation, the node verifies them again when the block is received. Consensus cannot rely on checks performed by other peers. Re-verification guarantees that the block genuinely carries votes from the participants it claims.

Finally, any proposer-specific votes are validated against the identity of the designated proposer for that slot. These messages have special significance in fork choice and must be signed by the correct validator at the correct time. This prevents impersonation of the proposer and preserves the integrity of block selection.

Together, these checks ensure that the block is internally coherent and cryptographically sound before the protocol considers the semantic meaning of its votes.

6.5.3 State Transition Processing: The Consensus Logic

A block that passes cryptographic verification may still contain votes that are irrelevant or outdated. Consensus depends not only on authenticity but also on how each vote relates to the current state of justification and finalization. The state transition function Υ therefore evaluates attestations semantically, determining whether they should influence fork choice and finality.

These checks are intentionally soft. An attestation that fails them does not invalidate the block; it is simply ignored during processing. This design allows proposers to include votes that may be slightly stale without risking rejection, while ensuring that only meaningful votes affect consensus.

6.5.3.1 Anchored in justified history

Finality advances as a chain of trust from one justified checkpoint to the next. A vote can only extend this chain if it begins from a checkpoint that is already recognized as justified in the current state. By enforcing this condition, the protocol guarantees that justification progresses along a single coherent history rather than jumping between competing forks or reverting to outdated views.

6.5.3.2 Eligible targets under structured finality

Even when anchored in justified history, not every future checkpoint is eligible to become justified next. Lean Consensus restricts admissible targets through a sparsification mechanism known as 3SF-mini. Instead of allowing votes for arbitrary future slots, the protocol permits targets only within a dense window near the finalized frontier and a carefully structured set of more distant checkpoints, as illustrated in Figure 6.6.

This structure serves two purposes. It limits the number of candidate links that nodes must track, reducing computational overhead, and it constrains long-range attacks by preventing validators from forming arbitrary justification paths far into the future. Finality must therefore advance through a controlled lattice of admissible checkpoints rather than an unrestricted timeline.

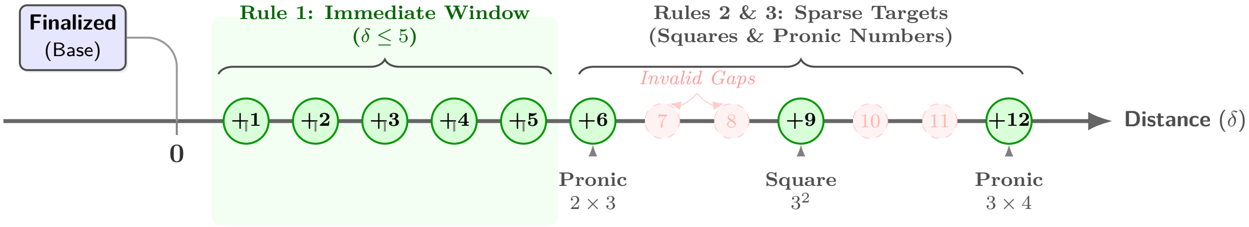


Figure 6.6: The 3SF-mini Admissibility Logic. The protocol enforces three rules for justification targets. (1) Any target within the Immediate Window ($\delta \leq 5$) is valid. Beyond that, targets must be either (2) a Perfect Square (e.g., $9, 16 \ldots$) or (3) a Pronic Number (e.g., $6, 12 \ldots$). All intermediate slots (shown in dashed red) are invalid targets.

6.5.3.3 Accumulating voting weight

When an attestation satisfies the semantic conditions above, its vote is applied to the corresponding source–target link. The protocol aggregates votes across all participating validators for that link. Once the accumulated vote count crosses the supermajority threshold

$$3 \cdot \text{votes} \geq 2 \cdot \text{total validators}, \tag{6.3}$$

the target checkpoint becomes justified. At that moment, the protocol records the new justified checkpoint and prepares the conditions under which finalization may advance.

6.5.4 Block-Building Filters

The final perspective is that of the proposer assembling a new block. From the proposer's viewpoint, attestations are resources competing for limited block space. Including every available vote would be wasteful and might even crowd out more valuable transactions. The proposer therefore selects aggregates that are both verifiable and useful to the protocol.

An attestation is attractive for inclusion only if other nodes will be able to verify it and if it contributes to advancing justification under the proposer's current view of the chain. Votes anchored to outdated checkpoints, or lacking valid proofs, provide no benefit and are typically omitted.

Efficiency also matters. Aggregated attestations may overlap in the validators they represent, as illustrated in Figure 6.7. Including redundant aggregates wastes space without increasing the recorded voting power. Proposers therefore prefer combinations that cover the largest number of distinct validators using the fewest aggregates. This selection process effectively packs the maximum amount of consensus information into the available block space, ensuring that each block carries a compact yet powerful summary of the network's agreement.

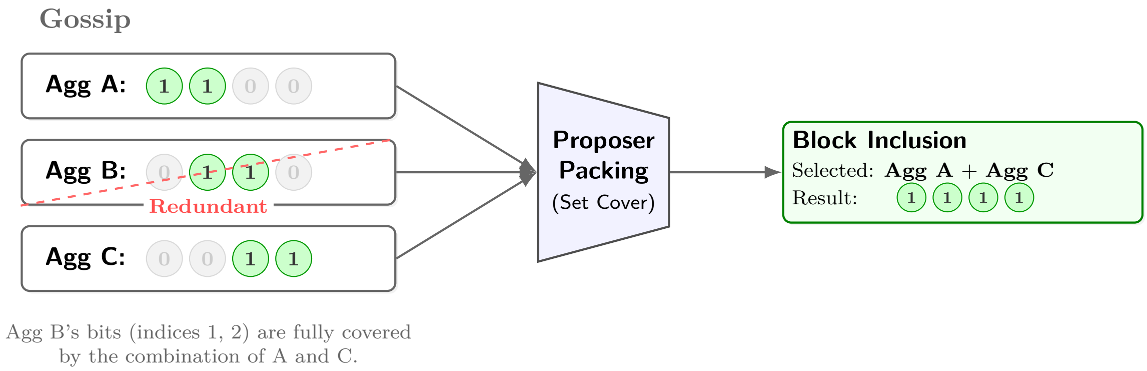


Figure 6.7: Proposer Efficiency Strategy. A proposer observes many overlapping aggregates in the gossip pool. To save block space, the proposer solves a set-cover problem: selecting the minimal subset of aggregates (here, A and C) that maximizes the total validator weight included, discarding redundant messages (like B).

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

[8] J. Iyengar and M. Thomson, "Rfc 9000: Quic: A udp-based multiplexed and secure transport," *Omtermet Emgomeeromg Task Force*, p. 4, 2021.