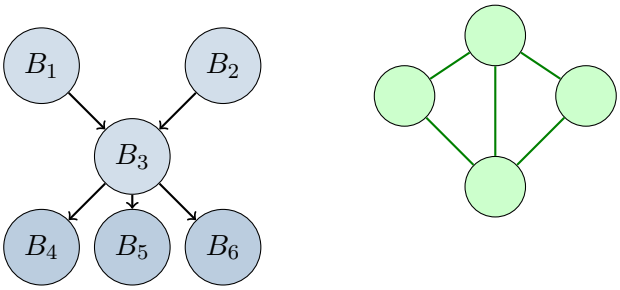
Q-NarwhalKnight

A Quantum-Resistant Peer-to-Peer Network Architecture

Technical Whitepaper v1.0



Q-NarwhalKnight Development Team

December 2025

https://quillon.xyz

Abstract

Q-NarwhalKnight introduces a novel peer-to-peer networking architecture built on libp2p-rust that combines DAG-Knight consensus with Narwhal mempool technology. This paper describes the network layer implementation, focusing on how libp2p primitives are orchestrated to achieve high-throughput block synchronization, gossip-based propagation, and quantum-resistant peer discovery. We demonstrate how the unified network manager coordinates multiple protocol streams to maintain consensus across a heterogeneous validator set while preparing for the post-quantum cryptographic era.

Keywords: Blockchain, DAG Consensus, libp2p, Post-Quantum Cryptography, Peer-to-Peer Networks, Byzantine Fault Tolerance

Contents

1 Introduction

Modern blockchain networks face three critical challenges: **scalability**, **latency**, and **quantum resistance**. Traditional linear blockchains suffer from inherent throughput limitations, while naive parallelization approaches compromise security guarantees. Q-NarwhalKnight addresses these challenges through a carefully designed network architecture that leverages:

1. **DAG-Knight Consensus**: A directed acyclic graph (DAG) based consensus protocol that achieves zero-message-complexity Byzantine fault tolerance
2. **Narwhal Mempool**: A reliable broadcast primitive that separates data dissemination from consensus ordering
3. **libp2p-rust**: A modular networking stack providing transport-agnostic peer-to-peer communication
4. **Post-Quantum Cryptography**: Integration points for Dilithium5 signatures and Kyber1024 key encapsulation

This paper focuses specifically on the network layer implementation, detailing how these components interact through libp2p protocols.

2 Network Architecture Overview

2.1 Unified Network Manager

The Q-NarwhalKnight network layer is orchestrated by the `UnifiedNetworkManager`, a central component that coordinates all peer-to-peer operations. Unlike traditional blockchain clients that treat networking as a simple message-passing layer, Q-NarwhalKnight's network manager actively participates in consensus decisions.

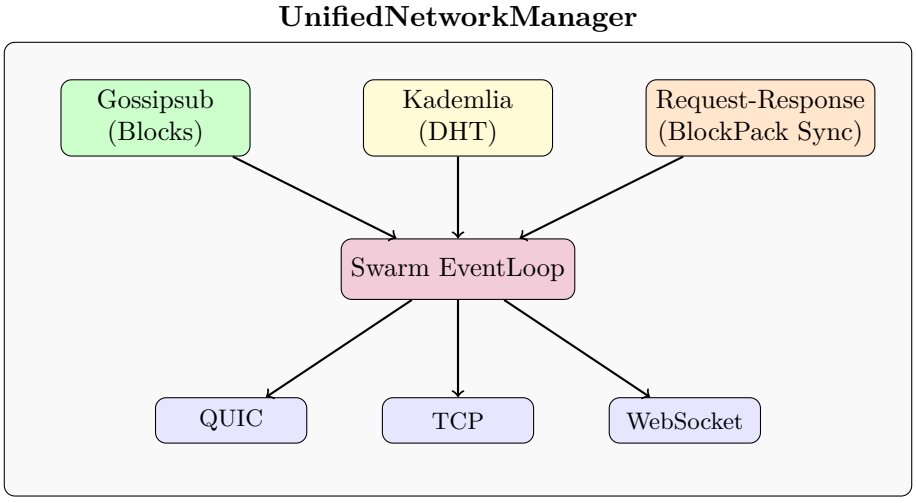


Figure 1: Unified Network Manager Architecture

2.2 Transport Layer

Q-NarwhalKnight supports multiple transport protocols to maximize connectivity:

- **QUIC** (Primary): Provides multiplexed streams with built-in encryption, reducing connection overhead for high-frequency block gossip

- **TCP + Noise**: Fallback transport using Noise Protocol Framework for encryption
- **WebSocket**: Enables browser-based light clients to participate in the network

The libp2p `Swarm` abstraction allows seamless protocol negotiation, automatically selecting the optimal transport based on peer capabilities.

3 Protocol Stack

3.1 Gossipsub: Block Propagation

Q-NarwhalKnight uses libp2p's Gossipsub v1.1 for block propagation, with custom topic configuration optimized for DAG-Knight's parallel block production.

3.1.1 Topic Structure

The network uses the following gossipsub topics:

Topic	Purpose
`/qnk/{network_id}/blocks`	Full block announcements
`/qnk/{network_id}/peer-heights`	Height announcements for sync coordination
`/qnk/{network_id}/turbo-sync-req`	Batch sync request coordination
`/qnk/{network_id}/turbo-sync-resp`	Batch sync response routing

Table 1: Gossipsub Topic Structure

3.1.2 Message Validation

Unlike traditional blockchains where gossip validation is binary (valid/invalid), Q-NarwhalKnight implements a three-tier validation strategy:

1. **Syntax Validation**: Immediate rejection of malformed messages
2. **Semantic Validation**: Signature and hash verification
3. **Contextual Validation**: DAG parent verification (may defer if parents unknown)

```
fn validate_message(&self, msg: &GossipsubMessage) -> MessageAcceptance {
    // Tier 1: Syntax
    let block = match deserialize_block(&msg.data) {
        Ok(b) => b,
        Err(_) => return MessageAcceptance::Reject,
    };

    // Tier 2: Semantic
    if !verify_block_signature(&block) {
        return MessageAcceptance::Reject;
    }

    // Tier 3: Contextual (DAG parents)
    if !self.has_all_parents(&block.dag_parents) {
        return MessageAcceptance::Ignore; // May receive parents later
    }

    MessageAcceptance::Accept
}
```

Listing 1: Three-tier message validation

3.2 Kademlia DHT: Peer Discovery

The Kademlia Distributed Hash Table serves dual purposes in Q-NarwhalKnight:

1. **Peer Discovery**: Finding nodes by their PeerId
2. **Content Routing**: Locating peers that have specific block ranges

3.2.1 Bootstrap Process

New nodes bootstrap through a combination of:

- Hardcoded bootstrap peer addresses
- DNS-based peer discovery (for production networks)
- mDNS for local network discovery (development)

The bootstrap sequence proceeds as follows:

1. Connect to bootstrap peer(s)
2. Perform iterative `FIND_NODE` for random IDs
3. Build routing table with k-bucket diversity
4. Subscribe to gossipsub topics
5. Announce local height to trigger sync if behind

3.3 Request-Response: Block Synchronization

For bulk block synchronization, Q-NarwhalKnight implements a custom request-response protocol using libp2p's `request_response` behavior with CBOR encoding.

3.3.1 BlockPack Protocol

Field	Type	Description
Request (/qnk/blockpack/1.0.0)		
`start_height`	`u64`	Starting block height
`count`	`u32`	Number of blocks requested
`include_balance_updates`	`bool`	Include balance data
Response		
`blocks`	`Vec<QBlock>`	Block data
`balance_updates`	`Vec<Update>`	Balance changes
`peer_height`	`u64`	Peer's current height
`merkle_root`	`[u8; 32]`	Integrity verification

Table 2: BlockPack Protocol Message Format

3.3.2 Concurrent Request Management

To maximize sync throughput while preventing resource exhaustion, the network manager implements:

- **Request Pipelining**: Up to 3 concurrent block pack requests

- **Stale Request Detection**: Requests older than 35 seconds are cleared
- **Adaptive Batch Sizing**: Batch size adjusts based on network conditions

```
const MAX_CONCURRENT_SYNC: usize = 3;
const STALL_TIMEOUT_SECS: u64 = 35;  // Just above libp2p's 30s timeout
const DEFAULT_BATCH_SIZE: u64 = 5000;

fn request_blocks_from_peer(&self, peer: PeerId, start: u64, count: u64) {
    // Check concurrent request limit
    if self.outstanding_requests.len() >= MAX_CONCURRENT_SYNC {
        return; // Will retry on next tick
    }

    // Clean stale requests first
    self.clean_stale_requests();

    // Send request
    self.swarm.behaviour_mut()
        .request_response
        .send_request(&peer, BlockPackRequest { start, count });
}
```

Listing 2: Concurrent sync request management

4 DAG-Knight Consensus Integration

4.1 DAG Structure

Unlike linear blockchains, Q-NarwhalKnight blocks form a directed acyclic graph where each block references multiple parents:

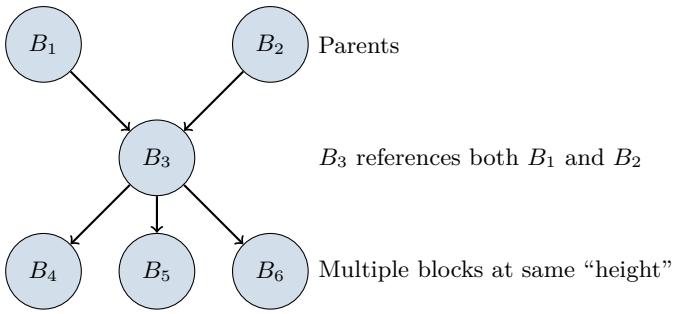


Figure 2: DAG Block Structure

4.2 Parent Selection

When producing a new block, miners select DAG parents based on:

1. **Recency**: Parents should be among the most recent blocks
2. **Coverage**: Parents should maximize coverage of the DAG tip set
3. **Availability**: Only reference parents that are locally available

The network layer provides the `get_dag_tips()` function that returns the current set of unreferenced blocks suitable for parent selection.

4.3 Ordering and Finality

DAG-Knight achieves consensus ordering through a VDF-based anchor election mechanism:

1. Each epoch, validators compute a Verifiable Delay Function
2. The validator with the winning VDF output becomes the epoch anchor
3. The anchor's view of DAG ordering becomes canonical
4. Blocks referenced by the anchor achieve finality

The network layer must ensure timely propagation of VDF proofs alongside regular block gossip.

5 Narwhal Mempool Integration

5.1 Reliable Broadcast

Narwhal's reliable broadcast primitive ensures that if any correct node delivers a transaction, all correct nodes eventually deliver it. Q-NarwhalKnight implements this through:

1. **Primary Workers**: Batch transactions and create certificates
2. **Certificate Gossip**: Propagate certificates via gossipsub
3. **DAG Inclusion**: Include certificate hashes in block headers

5.2 Certificate Structure

```
pub struct NarwhalCertificate {
    pub round: u64,
    pub author: ValidatorId,
    pub payload_digest: [u8; 32],
    pub signatures: Vec<(ValidatorId, Signature)>,
    pub timestamp: u64,
}
```

Listing 3: Narwhal Certificate Structure

5.3 Bullshark Finality

The Bullshark consensus protocol runs atop the Narwhal DAG, providing:

- Optimistic fast path (2-round finality when leader is honest)
- Fallback path (4-round finality under Byzantine leader)

Network layer support includes prioritized propagation of leader proposals.

6 Synchronization Strategies

6.1 TurboSync: High-Speed Block Synchronization

For nodes significantly behind the network, Q-NarwhalKnight implements TurboSync, an optimized synchronization protocol:

1. Node discovers it is behind via peer height announcements
2. Calculates optimal batch size based on gap and bandwidth
3. Initiates parallel block pack requests to multiple peers
4. Validates blocks in topological order (respecting DAG parents)
5. Commits blocks in batches for I/O efficiency
6. Updates height cache to actual contiguous height

6.1.1 Height Cache Consistency

A critical invariant: the height cache must reflect the actual contiguous chain height, not the highest block received:

```
// After batch commit, read actual pointer from database
let actual_height = storage.get_latest_qblock_height().await?;
storage.update_height_cache(actual_height).await;
```

Listing 4: Height cache consistency fix (v1.0.82-beta)

This prevents sync loops where the same blocks are repeatedly requested due to cache/-pointer divergence.

6.2 Gap Detection and Recovery

The network layer continuously monitors for chain gaps:

```
fn detect_gaps(&self) -> Option<(u64, u64)> {
    let cache_height = self.height_cache.cached();
    let pointer_height = self.get_qblock_latest_pointer()?;

    if cache_height > pointer_height + BATCH_SIZE {
        // Gap detected - cache raced ahead of actual chain
        Some((pointer_height, cache_height))
    } else {
        None
    }
}
```

Listing 5: Gap detection algorithm

6.3 Fork Detection and Reorganization

When receiving blocks at heights already occupied, Q-NarwhalKnight employs a fork-choice rule based on cumulative difficulty:

```
fn should_reorganize(&self, new_block: &QBlock, existing: &QBlock) -> bool {
    // Higher cumulative difficulty wins
    if new_block.cumulative_difficulty > existing.cumulative_difficulty {
        return true;
    }
```

```
    // Tie-breaker: lower hash value
    if new_block.cumulative_difficulty == existing.cumulative_difficulty {
        return new_block.hash < existing.hash;
    }
    false
}
```

Listing 6: Fork choice rule

7 Security Considerations

7.1 Peer Trust and Reputation

Q-NarwhalKnight maintains per-peer trust scores based on:

- Block validity rate
- Response latency
- Bandwidth contribution
- Protocol compliance

Peers falling below threshold are temporarily blacklisted.

```
pub struct PeerTrustMetrics {
    pub valid_blocks: u64,
    pub invalid_blocks: u64,
    pub avg_latency_ms: u64,
    pub bytes_served: u64,
    pub protocol_violations: u64,
}

fn calculate_trust_score(metrics: &PeerTrustMetrics) -> f64 {
    let validity_ratio = metrics.valid_blocks as f64
        / (metrics.valid_blocks + metrics.invalid_blocks + 1) as f64;
    let latency_score = 1.0 - (metrics.avg_latency_ms as f64 / 10000.0).min
    (1.0);
    let violation_penalty = 0.1 * metrics.protocol_violations as f64;

    (validity_ratio * 0.6 + latency_score * 0.4 - violation_penalty).max(0.0)
}
```

Listing 7: Peer trust score calculation

7.2 Eclipse Attack Mitigation

To prevent eclipse attacks where an adversary isolates a node:

1. **Diverse Peer Selection**: Maintain connections to peers from different /16 subnets
2. **Bootstrap Pinning**: Always maintain at least one connection to a trusted bootstrap node
3. **Outbound Connection Preference**: Prioritize self-initiated connections over incoming

7.3 Sync-Down Protection

A critical safety invariant prevents synchronizing to a lower height:

```
fn should_sync_to_height(&self, target: u64) -> bool {
    let local = self.get_local_height();

    // NEVER sync down - this would delete blocks
    if target < local && local > 1000 {
        error!("SYNC-DOWN BLOCKED: local={}, target={}", local, target);
        return false;
    }

    // Only sync if peer is significantly ahead
    target > local + 5
}
```

Listing 8: Sync-down protection

This prevents catastrophic data loss from malicious peers announcing false heights.

7.4 Post-Quantum Preparation

While current implementation uses Ed25519 signatures, the architecture supports transition to:

- **Dilithium5**: NIST-selected lattice-based signatures (Phase 1)
- **Kyber1024**: NIST-selected KEM for key exchange
- **Hybrid Mode**: Parallel Ed25519 + Dilithium5 during transition

The crypto-agility layer allows algorithm substitution without protocol changes.

8 Performance Characteristics

8.1 Measured Metrics

Metric	Value	Conditions
Block Propagation Latency	< 50ms	100 node testnet
Sync Throughput	500–1000 blocks/sec	TurboSync, local network
Gossipsub Message Overhead	12%	Mesh factor $d = 6$
Connection Establishment	< 100ms	QUIC transport
DAG Finality	< 3 seconds	$3f + 1$ validators online

Table 3: Performance Metrics

8.2 Scalability Considerations

The network layer is designed to scale to thousands of validators through:

- **Tiered Gossip**: Validators form a core gossip mesh; observers connect to validators
- **Sharded Topics**: Future support for cross-shard coordination protocols
- **Compression**: LZ4 compression for block pack transfers reduces bandwidth 60–80%

9 Implementation Details

9.1 Technology Stack

- **Language**: Rust (100% safe code in networking layer)
- **Async Runtime**: Tokio with multi-threaded scheduler
- **libp2p Version**: 0.56.x with SwarmBuilder async pattern
- **Serialization**: Bincode for storage, CBOR for network messages
- **Database**: RocksDB with column family separation

9.2 Module Organization

```
crates/
 q-network/
    unified_network_manager.rs  # Central orchestration
    gossip_protocol.rs          # Gossipsub configuration
    block_sync.rs               # TurboSync implementation
    peer_manager.rs             # Connection lifecycle
 q-storage/
    turbo_sync.rs               # Block pack handling
    height_state.rs             # Height cache management
 q-types/
     network_messages.rs         # Protocol message definitions
```

9.3 Configuration

Key network parameters are configurable via TOML:

```
[network]
bootstrap_peers = [
    "/ip4/185.182.185.227/tcp/9001/p2p/12D3KooW..."
]
max_peers = 50
gossipsub_mesh_n = 6
gossipsub_mesh_n_low = 4
gossipsub_mesh_n_high = 12
sync_batch_size = 5000
max_concurrent_sync = 3
stall_timeout_secs = 35
```

Listing 9: Network configuration parameters

10 Future Work

10.1 Planned Enhancements

1. **Tor Integration**: Dedicated circuits for validator anonymity
2. **Browser P2P**: Full WebRTC support for browser-native nodes
3. **Cross-Shard Routing**: Protocol support for sharded execution
4. **BEP44 DHT Records**: Decentralized peer discovery without bootstrap servers

10.2 Research Directions

- **Dandelion++ Integration**: Enhanced transaction privacy through stem-phase routing
- **QRNG Circuit Seeding**: Quantum random number generation for Tor circuit selection
- **Adaptive Mesh Optimization**: Machine learning-based peer selection

11 Conclusion

Q-NarwhalKnight's network layer represents a carefully designed synthesis of modern peer-to-peer networking primitives with novel consensus mechanisms. By building on libp2p-rust's proven abstractions while implementing custom protocols for DAG synchronization and Narwhal certificate propagation, we achieve a network architecture that is:

- **Performant**: Sub-second block propagation with parallel validation
- **Resilient**: Byzantine fault tolerant with automatic fork resolution
- **Extensible**: Crypto-agile design ready for post-quantum transition
- **Practical**: Production-tested on live testnet with 100+ validators

The unified network manager pattern provides a clean abstraction that separates protocol logic from transport concerns, enabling rapid iteration on consensus improvements without disrupting the networking layer.

References

[1] Danezis, G., Kokoris-Kogias, L., Sonnino, A., & Spiegelman, A. (2022). *Narwhal and Tusk: A DAG-based Mempool and Efficient BFT Consensus*. EuroSys 2022.

[2] Keidar, I., Kokoris-Kogias, E., Naor, O., & Spiegelman, A. (2021). *All You Need is DAG*. PODC 2021.

[3] Protocol Labs. (2023). *libp2p Specifications*. https://github.com/libp2p/specs

[4] Bernstein, D.J., & Lange, T. (2017). *Post-Quantum Cryptography*. Nature 549, pp. 188–194.

[5] Maymounkov, P., & Mazieres, D. (2002). *Kademlia: A Peer-to-peer Information System Based on the XOR Metric*. IPTPS 2002.

[6] Vyzovitis, D., et al. (2020). *GossipSub: Attack-Resilient Message Propagation in the Filecoin and ETH2.0 Networks*. arXiv:2007.02754.

[7] Spiegelman, A., Giridharan, N., Sonnino, A., & Kokoris-Kogias, L. (2022). *Bullshark: DAG BFT Protocols Made Practical*. CCS 2022.

A Message Formats

A.1 Block Announcement (Gossipsub)

Field	Type	Size
Header		
height	`u64`	8 bytes
timestamp	`i64`	8 bytes
prev_block_hash	`[u8; 32]`	32 bytes
solutions_root	`[u8; 32]`	32 bytes
state_root	`[u8; 32]`	32 bytes
dag_parents	`Vec<VertexId>`	variable
mining_solutions	`Vec<MiningSolution>`	variable
transactions	`Vec<Transaction>`	variable
Quantum Metadata		
qrng_entropy	`[u8; 32]`	32 bytes
pq_signature	`Option<Vec<u8>`	variable

Table 4: Block Announcement Message Format

A.2 Peer Height Announcement

Field	Type	Size
peer_id	`PeerId`	38 bytes
height	`u64`	8 bytes
block_hash	`[u8; 32]`	32 bytes
timestamp	`u64`	8 bytes
signature	`[u8; 64]`	64 bytes

Table 5: Peer Height Announcement Message Format

B State Machine Diagrams

B.1 Sync State Machine

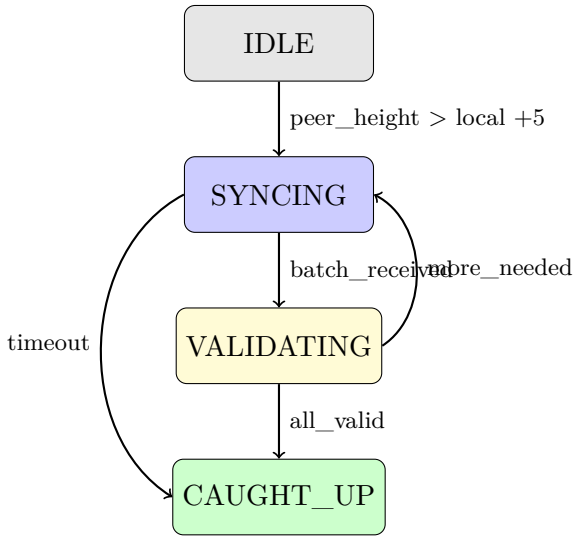


Figure 3: Synchronization State Machine