Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2024.0429000

RAVEN: Retention-Aware Verifiable Extension-block Network

EKATERINA BOCHVAROSKA[1] ALEKSANDAR TOŠIĆ[1,2],
[1] Faculty of Mathematics, Natural Sciences and Information Technologies, University of Primorska, 6000 Koper, Slovenia
[2] InnoRenew CoE, 6310 Izola, Slovenia

Corresponding author: Aleksandar Tošić (e-mail: aleksandar.tosic@upr.si).

ABSTRACT Centralized messaging platforms introduce structural vulnerabilities, including single points of failure and exposure to surveillance or censorship. Although blockchain-based messaging systems have been proposed to address some of these concerns, many existing approaches suffer from practical limitations due to on-chain storage.

This paper presents RAVEN, a retention-aware extension-block network that stores encrypted message data off-chain while anchoring block identifiers on-chain. In the proposed architecture, a parent chain's blocks are associated with an off-chain extension block EB_h, while the parent chain stores an anchor to the extension identifier $eid_h = \mathsf{H}(\mathrm{enc}(EB_h))$. Clients can therefore verify retrieved extension blocks against finalized parent-chain commitments while avoiding direct on-chain storage of complete message payloads. To improve efficiency of message retrieval under heterogeneous node retention, RAVEN introduces a retention-aware query decomposition algorithm that partitions client requests into sub-queries and distributes them across multiple candidate nodes in parallel.

The proposed retrieval mechanism was implemented and evaluated on a 24-node prototype network. Experimental results indicate low common-case retrieval latency and bounded additional delay under the evaluated concurrent workloads. A reference implementation of the protocol is provided and evaluated on a compute cluster. Our results show that on-chain message storage and retrieval are feasible.

INDEX TERMS blockchain, decentralized communication, censorship-resistant communication, peer-to-peer network,

I. INTRODUCTION

Messaging applications are essential to modern communication, enabling real-time interaction as part of the foundational systems that support social connections, economic coordination and political mobilization. Consequently, the architectural foundations of these platforms carry significant technical, ethical, and geopolitical weight. Historically, most messaging applications have been designed using centralized architectures. Prominent messaging services, including WhatsApp, Telegram, Signal, Slack, and Facebook Messenger, function within this centralized model. This model has been widely adopted because of its simplicity in terms of development, maintenance, and scalability. Nonetheless, although this model leverages effective engineering, coherent integration, and reliable uptime under ideal conditions, it simultaneously introduces structural vulnerabilities. Despite their convenience, centralized messaging systems are inherently prone to *single points of failure* as all message deliveries depend on operator-controlled servers, a deficiency long recognized in centralized network design [1]–[3]. These systems also exhibit *limited fault tolerance*, as failures within the central infrastructure cannot be routed around the network itself [2], [3]. Furthermore, they are vulnerable to *surveillance and censorship*, given that traffic must pass through central chokepoints where operators or entities can intercept, log, or filter communications [4], [5]. Additionally, there is a risk of *metadata leakage* [4], [6]–[8]. These risks are not incidental but inherent to the client–server paradigm underpinning most messaging services.

Furthermore, centralized platforms are inherently designed to maintain visibility over communication flows and retain auxiliary data such as user identities, timestamps, and device information. Even when end-to-end encryption protects message content, communication metadata such as sender identities, timestamps, and device information often remains visible to service providers and observers [6]. As a result, users remain exposed to metadata leakage that enables adversaries or providers to construct communication social graphs and infer behavioral patterns. Machado demonstrates that metadata

embedded in shared media files, including EXIF information such as GPS coordinates and timestamps, can further enable user tracking and behavioral reconstruction [9]. Such metadata enables adversaries to construct detailed social graphs and conduct behavioral profiling, which may be exploited for targeted advertising, or other forms of data-driven control. Consequently, protecting message content alone is insufficient to guarantee privacy in messaging applications.

Beyond technical risks, centralized services are also subject to external pressures that amplify these vulnerabilities. Platforms are frequently compelled to comply with governmental demands for data access or to restrict information during politically sensitive periods such as elections and protests. In China, deep-packet inspection systems are strategically deployed at border gateways to enforce censorship policies, underscoring how centralized control points facilitate large-scale information restriction [10].

In 2023 alone, 283 instances of deliberate internet shutdowns were documented across 39 countries, marking a 41% increase compared to the previous year. [11]

To overcome these limitations, significant efforts both in academic research and in practical deployments have focused on decentralization as an alternative architectural paradigm [12]–[15]. Recently popularized in the financial sector through Bitcoin [16], blockchain technology provides immutability, transparency, and distributed trust through cryptographic validation and consensus protocols. The immutable nature of blockchain records provides resilience against tampering or revision once data has been confirmed. In decentralized settings, this can support verifiable recording of references to off-chain data, although broader properties such as availability, metadata privacy, and censorship resistance still depend on protocol and network design.

In this paper, we propose a retrieval architecture that uses the blockchain ordering while integrating off-chain storage and a retrieval mechanism for extension blocks. The architecture is deployed as a layer on top of an existing blockchain, which continues to operate independently for its primary purposes. The technical focus of this work is the retrieval path: client requests for extension blocks are partitioned into sub-queries and distributed across multiple candidate nodes according to retention metadata.

The broader motivation for decentralized communication includes settings where centralized control is undesirable or risky. However, the present work is limited to the design and prototype evaluation of storage and retrieval behavior. More specifically, the evidence presented here supports cryptographic anchoring of off-chain data and prototype retrieval feasibility. It does not establish metadata privacy, adversarial resilience, network-level censorship resistance, or full end-to-end messaging guarantees. In particular, the cited literature on metadata privacy addresses a broader problem setting than the retrieval behavior evaluated here [8].

The main contributions of this work are summarized as follows:

- We describe an architecture in which encrypted message data is stored in off-chain extension blocks while the blockchain stores cryptographic references used for verification.
- We introduce a retention-aware query decomposition mechanism that partitions client requests into sub-queries and distributes them across multiple nodes in parallel.
- We design a distributed retrieval protocol that exploits heterogeneous node retention and supports fallback across candidate nodes during retrieval.

  We implement and evaluate the proposed retrieval mechanism on a 24-node network and analyze query latency, node participation, and retrieval behavior under concurrent workloads.

By building on an already operational blockchain network, the architecture uses the underlying ledger for ordering and integrity anchoring of off-chain data. The present study evaluates this design at prototype scale and focuses on retrieval efficiency rather than on all properties of a complete messaging system. In particular, integrity anchoring should not be read as evidence of availability, confidentiality, metadata privacy, or censorship resistance. This design keeps the main ledger lightweight while allowing clients to check consistency between returned extension blocks and on-chain commitments. In this design, blockchain anchoring supports integrity checking of retrieved data, but retrieval availability, metadata privacy, and censorship resistance depend on broader protocol and deployment assumptions that are not validated by the present experiments.

The remainder of this paper is organized as follows. Section II reviews related work. Section III presents the proposed methodology. Section IV describes the experimental setup and evaluation metrics. Section V presents the results and analysis, and Section VI concludes the paper.

II. RELATED WORKS

Early notable developments in this field include Chaum's Mix Networks from the 1980s [17], Onion Routing from the 1990s [18], P2P chat applications Jabber/XMPP [19], and experimental overlay GNUnet from the 2000s. [20] These technologies aimed to anonymize and distribute communication without the need for central intermediaries. Despite their innovative approach, these systems have failed to gain widespread adoption because of limited network effects, high maintenance costs for nodes, and usability tradeoffs. Early efforts aimed to replace centralized architectures with peer-to-peer communication, whereas recent approaches attempted to use blockchain for immutability and censorship resistance.

One of the earliest blockchain-based messaging proposals was introduced in 2018 [21], where a decentralized messenger was designed for communication in untrusted environments. The system relied on the Ethereum Whisper protocol to conceal recipient identities. However, the work did not evaluate performance metrics such as latency or throughput, leaving the practical feasibility of the design unclear.

In 2021, another blockchain-based messaging system was proposed to enhance security against centralized vulnerabil-

ities such as denial-of-service attacks [22]. Messages were stored as transactions within blocks and encrypted with recipients' public keys. The design leveraged Proof-of-Work consensus and SHA-256 signatures to guarantee integrity. However, the approach suffered from performance bottlenecks, as storing entire messages directly on-chain led to slow verification and high computational overhead. Subsequent works attempted to refine blockchain chat application with an emphasis on privacy, censorship resistance, and immutability [23]. Another approach was to adopt a decentralized application model on Ethereum, relying on end-to-end encryption and cryptographic replication across peers. In this model, the requirement that every node replicate all messages introduced scalability constraints. Researchers further explored Ethereum-based chat application utilizing smart contracts to automate message validation and delivery. Although trustless and transparent, the work highlighted front-end implementation while still overloading the main chain, resulting in scalability and cost limitations. [24]. Another recent proposal developed a private blockchain-based instant messaging platform [25]. Nevertheless, its application was limited to private settings, and it did not include quantitative assessments of the protocol's responsiveness, scalability, latency, and throughput. More recently, a 2025 study presented a peer-to-peer communication system that combined blockchain for authentication with direct node-to-node message exchange [26]. Specifically, the design incurs high transaction costs due to extensive on-chain functionality, while also showing limited throughput that constrains scalability. Moreover, resilience against Sybil and DDoS attacks remained untested, and performance evaluation was restricted to small-scale simulations, leaving broader empirical validation for future work. Collectively, these studies illustrate both the potential and the limitations inherent in blockchain-based messaging systems. Initial designs typically burdened the blockchain with complete message data, which degraded verification speed and inflated storage and computational cost. Few studies have identified the need for a second-layer solution to address the challenges of scalability and efficient data retrieval. Moreover, although many works explicitly acknowledge the importance of quantitative evaluation, none provide comprehensive empirical metrics such as latency, throughput, or resilience under adversarial conditions. Therefore, this lack of systematic evaluation leaves the true performance and practicality of these systems uncertain. Our proposed retrieval architecture addresses part of this design space by integrating blockchain transactions with references to off-chain extension blocks, thereby maintaining a lightweight ledger while supporting verifiability of stored data. The architecture uses query decomposition and parallel retrieval across multiple nodes to reduce latency in heterogeneous retention settings.

III. METHODOLOGY

The proposed protocol addresses low-latency access to encrypted message data that is stored off-chain and anchored on-chain. In contrast to systems that store complete messages directly on the blockchain, the proposed architecture stores compact cryptographic commitments on a parent chain, while the full encrypted message data is maintained in off-chain extension blocks. The main technical contribution is a retention-aware retrieval mechanism that decomposes a client request into sub-intervals and retrieves the corresponding extension blocks concurrently from candidate nodes according to their advertised retention metadata.

The methodology is organized as follows. First, the system architecture and data model are defined, including the encrypted-message pool, extension blocks, and parent-chain anchor records. Next, the write-path admission assumptions are stated. The retention model and network layer are then introduced. Finally, the section presents the parent-chain verification model, reorganization handling, query decomposition, query dispatch, and client-side validation.

A. SYSTEM OVERVIEW

The architecture follows a dual-layer design. The parent-chain layer provides ordering and cryptographic anchoring, whereas the off-chain layer stores encrypted message data in extension blocks. This separation keeps the parent chain lightweight while allowing clients to verify that retrieved off-chain data is consistent with commitments recorded on the parent chain.

The system consists of three logical components. First, the parent-chain layer contains ordered blocks and anchor records. In the base protocol evaluated in this work, each protocol-participating parent-chain height is associated with one extension-block anchor. In a native deployment, this anchor may be included directly by the block-producing node. In an adapter-based deployment over an existing parent chain, the same abstraction may be implemented by an anchor transaction, contract call, memo field, or equivalent chain-specific record. Second, the storage and retrieval layer consists of nodes that retain extension blocks according to heterogeneous retention policies. Third, the client layer issues interval queries, retrieves extension blocks, and verifies the returned data against parent-chain commitments.

The network follows an on-demand access model. When issuing a query, a client obtains metadata about available nodes, including node type, advertised retention window, and retrieval endpoint. Based on this metadata, the client decomposes the requested interval into one or more sub-queries and dispatches them to candidate nodes. Retention metadata is used as routing information rather than as a proof of availability. Consequently, blockchain anchoring provides integrity and ordering verification for retrieved data, while availability, write-path spam resistance, metadata privacy, and adversarial robustness depend on broader deployment assumptions that are not fully validated by the present experiments.

B. ENCRYPTED-MESSAGE POOL AND EXTENSION-BLOCK CREATION

Before messages are anchored, nodes maintain a local encrypted-message pool. This pool is analogous to a transaction mempool in blockchain systems. Users submit encrypted

message envelopes to one or more nodes. Nodes apply a write-admission policy, discard invalid envelopes, and gossip valid encrypted envelopes to their peers.

Let M denote an encrypted message envelope. A valid envelope may contain fields such as a protocol version, recipient identifier or routing tag, ciphertext, timestamp or epoch, and an anti-spam cost signal. The exact envelope structure is deployment-specific, but nodes only admit envelopes that satisfy the admission predicate defined in Subsection III-C.

When a node proposes the next protocol-participating parent-chain block at height h, it selects a finite set of valid pending encrypted messages from its local pool:

$$\mathcal{T}_h = \{M_1, M_2, \ldots, M_{q_h}\}. \tag{1}$$

The proposer then constructs an off-chain extension block

$$EB_h = (v_h, \mathcal{T}_h, \mu_h), \tag{2}$$

where v_h is the extension-block format version, $\mathcal{T}_h$ is the set of encrypted message envelopes included at height h, and μ_h denotes auxiliary metadata such as message count, payload size, encoding information, or retention class.

The extension identifier is defined as the cryptographic hash of a canonical serialization of the extension block:

$$eid_h = \mathsf{H}(\mathrm{enc}(EB_h)), \tag{3}$$

where H is a collision-resistant hash function and $\mathrm{enc}(\cdot)$ denotes deterministic canonical encoding. Thus, eid_h is a cryptographic commitment to the full contents of EB_h. If a malicious node modifies any part of the extension block, the recomputed identifier changes and no longer matches the parent-chain anchor.

The proposed parent-chain block at height h contains an anchor record

$$AR_h = (protocolTag, eid_h), \tag{4}$$

or an equivalent chain-specific representation that commits to eid_h. The protocol does not require the parent-chain block header itself to be modified. Depending on the parent chain, the anchor may be represented as transaction metadata, contract calldata, a memo field, an application-specific transaction, an `OP_RETURN`-like construction, or, in a native implementation, a dedicated anchor field.

In the base protocol, there is one extension anchor per protocol-participating parent-chain height. If no encrypted messages are selected for a height, the proposer may still create and anchor an empty extension block. An empty extension block is still a valid extension block. This avoids ambiguity between a height with no messages and a height for which extension data is missing.

The parent-chain block determines the canonical ordering of extension blocks. The extension block does not need to include the parent-chain block hash. Instead, the parent-chain block points to eid_h, and eid_h commits to the extension-block contents. This removes a bidirectional dependency between the parent-chain block hash and the extension-block hash. As a result, if a parent-chain reorganization occurs, the extension block itself does not change; only its anchor status changes.

Algorithm 1 Extension-Block Construction by a Proposing Node

Require: Pending encrypted-message pool $\mathcal{P}$, maximum extension-block size $B_{\max}$, parent-chain height h
Ensure: Proposed parent-chain block containing anchor AR_h and off-chain extension block EB_h

$\mathcal{T}_h \leftarrow \varnothing$
$size \leftarrow 0$
for all $M \in \mathcal{P}$ **do**
  if $\mathrm{Admit}(M)$ and $size + |M| \le B_{\max}$ **then**
    $\mathcal{T}_h \leftarrow \mathcal{T}_h \cup \{M\}$
    $size \leftarrow size + |M|$
  end if
end for
$EB_h \leftarrow (v_h, \mathcal{T}_h, \mu_h)$
$eid_h \leftarrow \mathsf{H}(\mathrm{enc}(EB_h))$
$AR_h \leftarrow (protocolTag, eid_h)$
include AR_h in the proposed parent-chain block at height h
propagate EB_h to participating storage nodes
return (AR_h, EB_h)

The ordering policy used to select messages from the pending pool $\mathcal{P}$ is deployment-specific and may depend on arrival order, fees, proof-of-work strength, payload size, other local policies. The retrieval protocol requires only that selected messages satisfy the admission predicate and that the resulting extension block is committed through the anchored identifier.

C. WRITE-PATH ADMISSION AND SPAM RESISTANCE

The proposed retrieval mechanism assumes that encrypted messages admitted into extension blocks satisfy a deployment-specific write-admission policy. This paper does not define a universal tokenomic mechanism, since the architecture is intended to remain compatible with different parent chains and deployment settings. Nevertheless, any open deployment must impose some cost or admission rule on message submission, because extension-block storage and parent-chain anchor space are finite resources.

At a minimum, a message envelope must satisfy syntactic validity, freshness, and size constraints before it is included in an extension block. A generic admission predicate can be written as

$$\begin{aligned} \mathrm{Admit}(M) \iff\ & \mathrm{WellFormed}(M) \\ & \wedge\ |M| \le M_{\max} \\ & \wedge\ \mathrm{Fresh}(M) \\ & \wedge\ \mathrm{CostValid}(M, \Gamma), \end{aligned} \tag{5}$$

where $M_{\max}$ is the maximum message size and Γ denotes the deployment-specific admission context. The predicate $\mathrm{CostValid}$ may be instantiated using native transaction fees, a smart-contract deposit, payment to a protocol-defined address,

Algorithm 2 Generic Write-Admission Check

Require: Message envelope M, admission context Γ, maximum message size $M_{\max}$
Ensure: Accept or Reject

```
if ¬ WellFormed(M) then
    return Reject
end if
if |M| > M_max then
    return Reject
end if
if ¬ Fresh(M) then
    return Reject
end if
if ¬ CostValid(M, Γ) then
    return Reject
end if
return Accept
```

a message-bound proof-of-work stamp, or a hybrid mechanism.

For a proof-of-work-based deployment, the stamp may be bound to the message contents rather than to a sender identity:

$$\mathsf{H}(\mathit{networkID} \parallel \mathit{epoch} \parallel \mathsf{H}(M) \parallel |M| \parallel \mathit{nonce}) < \mathit{target}(|M|, \rho), \tag{6}$$

where *networkID* prevents cross-network replay, *epoch* limits reuse across time windows, $|M|$ binds the cost to message size, and ρ denotes recent extension-block occupancy or congestion. This construction prevents an adversary from avoiding the write cost merely by generating new public keys, because the cost is attached to each submitted message and its size.

The concrete economic or anti-spam mechanism is outside the scope of the present evaluation. The role of this subsection is to make explicit that the retrieval protocol does not by itself solve write-path spam. Instead, it assumes that nodes only propagate, store, and anchor encrypted messages that satisfy the chosen write-admission policy.

D. RETENTION MODEL

The retention model defines which extension blocks are stored and served by each node. A regular node stores only a suffix of recent extension blocks, while an archive node stores the complete extension-block history.

Let tip denote the current canonical parent-chain tip known to the client. A regular node n with retention window R_n stores extension blocks whose anchor heights lie in

$$[tip - R_n + 1, tip]. \tag{7}$$

Define the lower retention bound of a regular node as

$$\ell_n = tip - R_n + 1. \tag{8}$$

Archive nodes are treated as having $\ell_n = 1$. A node n covers the interval $I = [a, b]$ if and only if

$$I \subseteq [\ell_n, tip]. \tag{9}$$

Equivalently, node n can serve a request interval $[s, e]$ when

$$\ell_n \le s \quad \text{and} \quad e \le tip. \tag{10}$$

If a requested interval is not covered by any regular node, the client falls back to archive nodes when such nodes are available. If no candidate node covers the interval, the interval is considered unavailable. Retention metadata is therefore a routing indication rather than a proof that the data is actually present. False positives can occur if a node advertises a retention window but has pruned, lost, or failed to serve some extension blocks. The query dispatcher handles such cases through timeout-bounded fallback across candidate nodes, as described in Subsection III-I.

In the evaluated prototype, node retention capacities are assigned according to a **Pareto distribution**. This distribution captures scenarios in which a small subset of nodes manages a disproportionately large part of the stored data, reflecting the well-known 80/20 principle. This choice reflects the heterogeneity commonly observed in decentralized networks. The design is inspired by the **Tor network**, where nodes exhibit diverse capacities in terms of bandwidth and uptime, and where a small number of high-capacity relays handle a substantial fraction of the traffic [27].

E. NETWORK LAYER

Nodes communicate using two complementary communication channels. First, a WebSocket-based peer-to-peer channel is used in the prototype for message propagation, block synchronization, anchor propagation, and extension-block dissemination. Users submit encrypted message envelopes to one or more nodes. Nodes that accept the envelopes according to the write-admission policy propagate them to peers and store them temporarily in their local encrypted-message pools.

When a proposing node creates a parent-chain block, it selects encrypted messages from its pool, constructs the corresponding extension block EB_h, computes eid_h, and includes the anchor record $AR_h = (protocolTag, eid_h)$ in the proposed parent-chain block. The parent-chain block and the corresponding extension block are then propagated to peers. Nodes receiving the broadcast verify that

$$eid_h = \mathsf{H}(\mathrm{enc}(EB_h)) \tag{11}$$

before storing EB_h according to their retention policies. If a node detects that it is missing intermediate anchors or extension blocks, it requests the missing data from peers using a backfill procedure.

Second, a lightweight HTTP interface is used for on-demand retrieval during client queries. Each node exposes a query endpoint that accepts requests for a canonical parent-chain interval:

$$[start, end].$$

Upon receiving a request, the node retrieves the locally stored extension blocks and anchor bindings for the requested interval and returns them as a serialized response. This separation between propagation and retrieval simplifies the architecture

and allows the experimental evaluation to focus on distributed query behavior rather than on transport-specific details.

F. PARENT-CHAIN VERIFICATION MODEL

The protocol is implemented over a parent chain that provides ordering and cryptographic anchoring. However, the protocol does not require changes to the parent chain consensus rules or block-header format. Instead, it requires a chain-observation interface capable of exposing authenticated canonical-chain information to clients and participating nodes.

In secure client mode, clients verify the parent chain as light clients. The parent-chain adapter is modeled as

$$\begin{aligned}\mathcal{I}_{chain} = (\,&\mathrm{Head}, \mathrm{Headers},\\ &\mathrm{Parent}, \mathrm{BlockAtHeight},\\ &\mathrm{AnchorInBlock}, \mathrm{VerifyAnchor},\\ &\mathrm{IsCanonical}, \mathrm{IsFinal})\end{aligned}$$

where $\mathrm{Head}()$ returns the observed canonical tip, $\mathrm{Headers}(s, e)$ returns the parent-chain headers for an interval, $\mathrm{Parent}(b)$ returns the parent of block b, $\mathrm{BlockAtHeight}(h)$ returns the canonical block at height h, $\mathrm{AnchorInBlock}(b)$ returns the extension anchor contained in block b, $\mathrm{VerifyAnchor}(AR_h, \Pi_h, b_h)$ verifies that anchor record AR_h is included in parent-chain block b_h, $\mathrm{IsCanonical}(b_h)$ determines whether b_h belongs to the canonical chain, and $\mathrm{IsFinal}(b_h)$ applies the chain-specific finality rule.

For chains with transaction Merkle roots in block headers, Π_h may be a transaction-inclusion proof. For chains with smart contracts, it may be a receipt or state proof. For other chains, it may be any authenticated proof that the anchor record is included in a finalized parent-chain block. The exact proof format is chain-specific, but the verification condition is the same: the client must verify that the anchor record committing to eid_h belongs to the canonical finalized parent chain.

Let C denote the currently observed canonical parent chain. Let $\mathrm{Anchor}_C(eid_h, b_h)$ be true if canonical block $b_h \in C$ contains an authenticated anchor record committing to eid_h. Let $\mathrm{Final}_C(b_h)$ denote the finality predicate of the parent chain. For probabilistic-finality chains, this predicate may be defined using a confirmation threshold k:

$$\mathrm{Final}_C(b_h) \iff depth_C(b_h) \geq k. \tag{12}$$

For chains with explicit finality, the predicate may instead refer to a finalized checkpoint or an equivalent chain-specific primitive.

An extension block is accepted as final with respect to the canonical parent chain only if its content hash matches the anchored identifier and the corresponding anchor is both canonical and final:

$$\begin{aligned}\mathrm{Valid}_C(EB_h) \iff\ & eid_h = \mathsf{H}(\mathrm{enc}(EB_h))\\ & \wedge\ \exists b_h \in C : \mathrm{Anchor}_C(eid_h, b_h)\\ & \wedge\ \mathrm{Final}_C(b_h).\end{aligned} \tag{13}$$

If light-client verification is unavailable or too expensive for a specific deployment, the client may use a weaker multi-peer synchronization mode. In that mode, the client queries several independently selected peers for parent-chain tips and anchor records and accepts a view only when a threshold of peers returns the same finalized checkpoint and anchor set. This mode can improve robustness against isolated faulty nodes, but it does not provide the same security as direct light-client verification and remains vulnerable when the sampled peers are adversarial or Sybil-controlled.

G. REORGANIZATION HANDLING

The protocol inherits the reorganization behavior of the underlying parent chain. If a parent-chain block containing an extension anchor is removed from the canonical chain, the corresponding anchor binding becomes non-canonical. The extension block itself is not modified, because its identity is content-addressed by eid_h. A reorganization changes the anchor bindings, not the extension-block contents.

The state of an anchor binding is one of

$$status \in \{\mathsf{Pending}, \mathsf{Confirmed}, \mathsf{Final}, \mathsf{Orphaned}\}. \tag{14}$$

A binding is Pending if the anchor record is known but not yet included in the observed canonical chain. It is Confirmed if the anchor appears in the canonical chain but has not yet satisfied the finality predicate. It is Final if the anchor appears in the canonical chain and satisfies the finality predicate. It is Orphaned if the anchor appeared in a block that was later removed by a reorganization.

When a newly observed head does not extend the previously observed head, a node identifies the common ancestor, marks anchors in the removed branch as orphaned, scans the replacement branch for anchor records, and updates anchor bindings accordingly. If the same eid_h appears again in the replacement canonical branch, the corresponding extension block can be rebound to the new canonical anchor location. If it does not appear, the extension block remains locally stored but non-final until it is re-anchored, re-included in a later valid extension block, or discarded according to the node's local orphan-cache policy.

Reorganizations may therefore delay message finality, but they do not cause clients to accept extension blocks anchored only in orphaned parent-chain blocks. A client accepts data only after verifying the final parent-chain anchor and the extension-block content hash.

H. QUERY DECOMPOSITION ALGORITHM

A central contribution of the protocol is a retention-aware query decomposition mechanism that partitions a client request into sub-intervals according to node coverage boundaries. In contrast to fixed-size chunking approaches, the implemented algorithm does not divide the query range into equal segments. Instead, it uses the retention metadata of currently available nodes and creates a new segment only when the set of nodes capable of serving the queried height changes. This produces a compact decomposition consistent with the storage distribution of the network.

Algorithm 3 Reorganization-Aware Anchor Tracking

Require: Previous head *oldHead*, new head *newHead*, anchor index *AI*, extension-block store *ES*, chain adapter $\mathcal{I}_{chain}$

Ensure: Updated anchor index *AI*

if $\mathcal{I}_{chain}.\mathrm{Parent}(newHead) = oldHead$ **then**
  $AddedBranch \leftarrow \{newHead\}$
  $RemovedBranch \leftarrow \emptyset$
else
  $c \leftarrow \mathrm{FindCommonAncestor}(oldHead, newHead)$
  $RemovedBranch \leftarrow \mathrm{BlocksBetween}(oldHead, c)$
  $AddedBranch \leftarrow \mathrm{BlocksBetween}(c, newHead)$
end if
for all $b \in RemovedBranch$ **do**
  $AR \leftarrow \mathcal{I}_{chain}.\mathrm{AnchorInBlock}(b)$
  if $AR \neq \perp$ **then**
    $\mathrm{MarkBinding}(AI[AR.eid], b.hash, \mathsf{Orphaned})$
  end if
end for
for all $b \in AddedBranch$ **do**
  $AR \leftarrow \mathcal{I}_{chain}.\mathrm{AnchorInBlock}(b)$
  if $AR \neq \perp$ and $\mathcal{I}_{chain}.\mathrm{VerifyAnchor}(AR, AR.\Pi, b)$ **then**
    $binding \leftarrow (AR.eid, b.height, b.hash, AR.txid, AR.\Pi, \mathsf{Confirmed})$
    $\mathrm{AddBinding}(AI[AR.eid], binding)$
  end if
end for
for all $eid \in AI$ **do**
  for all $binding \in AI[eid]$ **do**
    if $\neg\mathcal{I}_{chain}.\mathrm{IsCanonical}(binding.blockHash)$ **then**
      $binding.status \leftarrow \mathsf{Orphaned}$
    else if $\mathcal{I}_{chain}.\mathrm{IsFinal}(binding.blockHash)$ **then**
      $binding.status \leftarrow \mathsf{Final}$
    else
      $binding.status \leftarrow \mathsf{Confirmed}$
    end if
  end for
end for
for all $eid \in ES$ **do**
  if $AI[eid]$ contains no Confirmed or Final binding **then**
    $\mathrm{RetainInOrphanCache}(ES[eid])$
  end if
end for
return *AI*

Let the client request be a range $Q = [s, e]$, where

$$1 \le s \le e \le tip,$$

and let the available nodes be

$$\mathcal{N} = \{n_1, n_2, \ldots, n_k\}.$$

The node list is sorted by increasing retention, meaning that nodes with smaller retained history windows appear first and archive nodes appear last. For regular nodes, this corresponds to non-increasing lower bounds:

$$L_1 \ge L_2 \ge \cdots \ge L_p,$$

where p is the number of regular nodes and $L_j = \ell_{n_j}$.

For any interval $I = [a, b] \subseteq [s, e]$, define the candidate set

$$\mathcal{C}(I) = \{n \in \mathcal{N} : I \subseteq [\ell_n, tip]\}.$$

The splitter processes the requested interval from right to left, beginning at height e. At each step, it determines the first regular node whose lower bound does not exceed the current endpoint. This node marks the next retention boundary. The algorithm then creates the largest suffix interval that can be served by the current candidate set and moves leftward until the entire request is covered. If no regular node can serve the remaining prefix, the residual interval is assigned to archive nodes.

Unlike fixed chunking, the number of generated sub-queries depends not on the absolute query length alone, but on the number of *retention boundaries* crossed by the request. If a query lies entirely within a region covered by the same node set, it remains a single sub-query even if its range is relatively large. Conversely, a shorter query may be split if it crosses several lower-bound transitions. This aligns the algorithm with the storage topology of the network and avoids unnecessary fragmentation.

The decomposition step returns a sequence of interval assignments

$$\mathcal{A} = \{(I_1, \mathcal{C}_1), (I_2, \mathcal{C}_2), \ldots, (I_m, \mathcal{C}_m)\},$$

where each $I_t \subseteq [s, e]$, the intervals are contiguous and non-overlapping, and

$$\bigcup_{t=1}^{m} I_t = [s, e].$$

Each interval is dispatched independently, enabling *inter-subquery parallelism*. Within each interval assignment, the dispatcher maintains a candidate set of nodes and attempts retrieval until one node returns a valid response or all candidates are exhausted.

The practical execution model consists of three stages, as illustrated in Figure 1. First, the splitter derives interval assignments from retention metadata. Second, the dispatcher asynchronously launches one retrieval task per interval. Third, within each task, candidate nodes are shuffled to avoid deterministic node bias, and the first valid response is accepted. Failures trigger retries on the remaining candidates. Once all interval tasks complete, the returned extension blocks are merged and sorted by canonical parent-chain height to reconstruct the original client request.

For an interval assignment $I_t = [a_t, b_t]$, the client accepts a returned sequence

$$\widehat{E}_t = (EB_{a_t}, EB_{a_t+1}, \ldots, EB_{b_t})$$

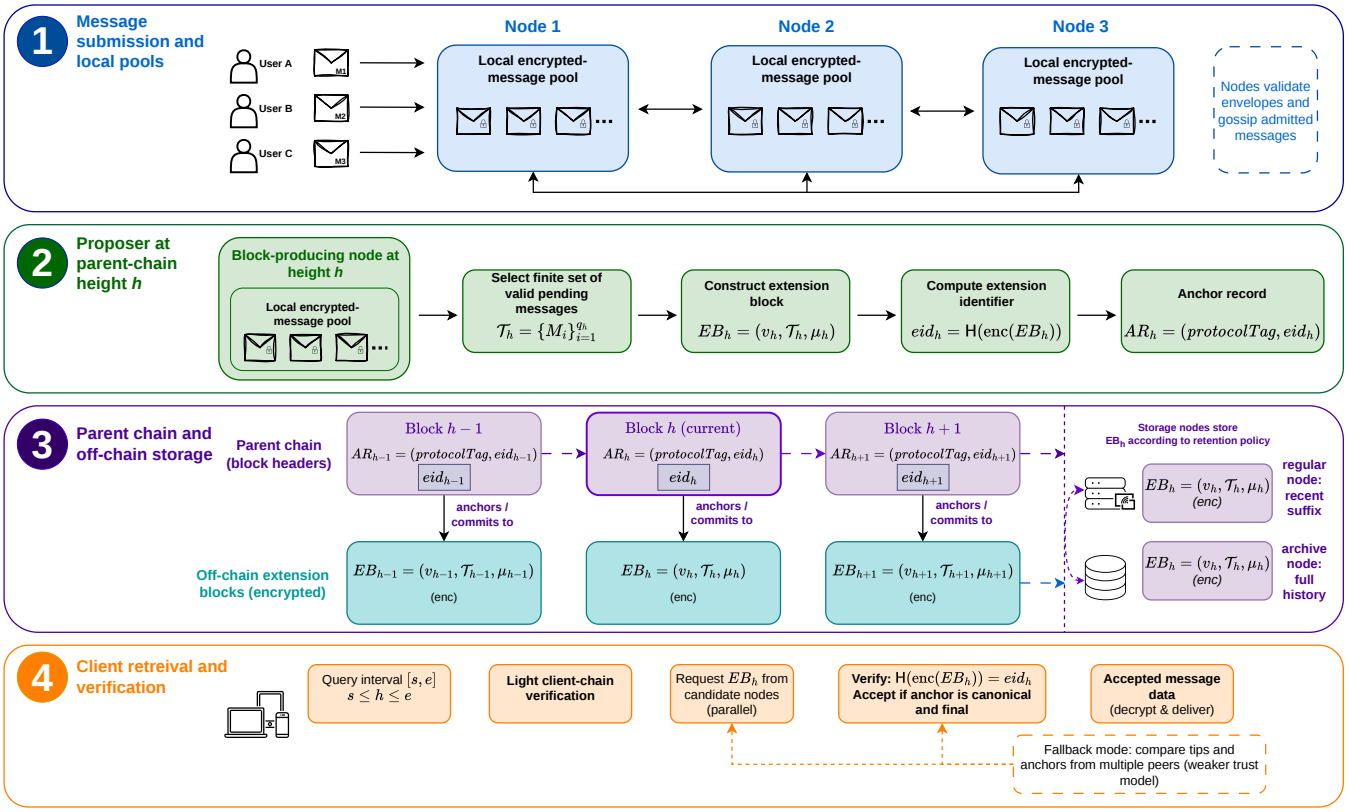


FIGURE 1. **System architecture and verification flow of the proposed protocol. Users submit encrypted message envelopes to local node pools. A proposing node selects valid pending messages, constructs the extension block EB_h, computes $eid_h = \mathsf{H}(\mathrm{enc}(EB_h))$, and anchors eid_h in the parent-chain block at height h. Storage nodes retain extension blocks according to their retention policies, while clients retrieve extension blocks from candidate nodes and verify them against finalized parent-chain anchors.**

only after checking consistency with the corresponding parent-chain commitments. The reconstructed response is therefore

$$\widehat{E}(Q) = \mathrm{sort}_h\left(\bigcup_{t=1}^{m} \widehat{E}_t\right).$$

This verification step provides integrity anchoring for retrieved data, but it does not by itself establish availability, honest serving behavior, write-path spam resistance, or metadata privacy.

I. QUERY DISPATCHER

After decomposition, each interval assignment is submitted to the query dispatcher. The dispatcher launches asynchronous retrieval tasks, one per interval, using the candidate set provided by the splitter. For each interval, candidate nodes are randomly shuffled before execution so that repeated experiments do not systematically prefer a fixed node order.

The dispatcher sends an HTTP request to the first candidate node. A response is considered valid only if it contains the requested extension blocks and corresponding anchor bindings, and if these objects pass the client-side validation procedure. If the node times out, refuses the connection, returns an incomplete response, or returns data that fails verification, the dispatcher retries the same interval on the next candidate node. This process continues until a valid response is obtained or all candidates are exhausted.

Thus, advertised retention metadata is not treated as an availability guarantee. It is only used to construct candidate sets. False positives in retention metadata are handled operationally through timeout-bounded fallback. A sub-query is considered unavailable only when all candidates for its interval fail.

Algorithm 5 detects both corrupted and incomplete data. Corruption is detected when an extension block does not hash to its anchored identifier. Omission is detected because the parent chain defines one expected extension anchor per protocol-participating height in the requested interval. If an expected extension block is missing, the dispatcher retries the interval against another candidate node.

For each sub-query, the implementation records the interval bounds, elapsed time, attempted nodes, and successful serving node. After all asynchronous tasks terminate, the returned extension blocks are aggregated and sorted by canonical parent-chain height to reconstruct the original requested interval.

J. CLIENT LAYER

Clients are intentionally lightweight and focus on querying and verifying data rather than participating in consensus or storing the full extension-block history. First, a client

Algorithm 4 Retention-Aware Greedy Query Decomposition

Require: Query interval $[s, e]$, current chain tip tip, node list $\mathcal{N}$ sorted by increasing retention, with archive nodes placed last
Ensure: Assignment set $\mathcal{A}$ of interval–candidate-set pairs

$\mathcal{A} \leftarrow \varnothing$
if $\mathcal{N} = \varnothing$ **then**
  return $\mathcal{A}$
end if
$s \leftarrow \max(1, s)$
$e \leftarrow \min(tip, e)$
if $e < s$ **then**
  return $\mathcal{A}$
end if
identify the prefix of regular nodes in $\mathcal{N}$ and let its size be p
$\mathcal{N}_{archive} \leftarrow \mathcal{N} \setminus \{n_1, \ldots, n_p\}$
if $p = 0$ **then**
  return $\{([s, e], \mathcal{N})\}$ ▷ all available nodes are archives
end if
extract lower bounds $L_1, \ldots, L_p$ from the regular nodes
$currentEnd \leftarrow e$
$j \leftarrow 1$
while $currentEnd \geq s$ **do**
  while $j \leq p$ and $L_j > currentEnd$ **do**
    $j \leftarrow j + 1$
  end while
  if $j > p$ **then**
    if $\mathcal{N}_{archive} \neq \varnothing$ **then**
      append $([s, currentEnd], \mathcal{N}_{archive})$ to $\mathcal{A}$
    else
      append $([s, currentEnd], \varnothing)$ to $\mathcal{A}$
    end if
    break
  end if
  $segmentStart \leftarrow \max(s, L_j)$
  append $([segmentStart, currentEnd], \{n_j, n_{j+1}, \ldots, n_k\})$ to $\mathcal{A}$
  $currentEnd \leftarrow segmentStart - 1$
end while
reverse $\mathcal{A}$ to restore ascending interval order
merge adjacent assignments with identical candidate sets
return $\mathcal{A}$

Algorithm 5 Client Validation of Retrieved Anchor Interval

Require: Requested interval $[s, e]$, retrieved extension blocks EB, chain adapter $\mathcal{I}_{chain}$
Ensure: Accept, Incomplete, or Reject

$Missing \leftarrow \varnothing$
for $h \leftarrow s$ to e **do**
  $b_h \leftarrow \mathcal{I}_{chain}.\mathrm{BlockAtHeight}(h)$
  $AR_h \leftarrow \mathcal{I}_{chain}.\mathrm{AnchorInBlock}(b_h)$
  if $AR_h = \perp$ **then**
    return Reject
  end if
  if $\neg\mathcal{I}_{chain}.\mathrm{VerifyAnchor}(AR_h, AR_h.\Pi, b_h)$ **then**
    return Reject
  end if
  if $\neg\mathcal{I}_{chain}.\mathrm{IsFinal}(b_h)$ **then**
    return Reject
  end if
  if EB_h was not returned **then**
    $Missing \leftarrow Missing \cup \{h\}$
  else
    if $\mathsf{H}(\mathrm{enc}(EB_h)) \neq AR_h.eid$ **then**
      return Reject
    end if
  end if
end for
if $Missing \neq \varnothing$ **then**
  return Incomplete
end if
return Accept

obtains a current local view of available nodes, including node type, advertised retention window, and endpoint information. Second, the client submits a range of canonical parent-chain heights corresponding to the messages it intends to retrieve.

Regular nodes prune older extension blocks and corresponding index entries while retaining only recent ones. Consequently, clients may encounter unavailable data during long-range queries. The protocol mitigates this by using retention metadata to route recent queries initially to regular nodes, while relying on archive nodes only when older data is necessary. In this way, recent queries can be served with lower storage overhead, while historical queries remain possible when archive coverage exists.

In secure mode, clients verify the parent chain as light clients. The client obtains parent-chain headers and authenticated anchor-inclusion proofs, verifies that the relevant anchor records are included in finalized canonical parent-chain blocks, retrieves the corresponding extension blocks, and verifies each extension block against the identifier committed by its parent-chain anchor. This mode prevents a retrieval node from modifying extension-block contents or silently omitting expected extension blocks without detection.

In the evaluated prototype, clients obtain the current chain tip from participating nodes before issuing interval requests. This is sufficient for evaluating retrieval behavior, but it is weaker than full parent-chain light-client verification. A production deployment should use light-client verification as the default mode. When this is impractical due to chain size, bandwidth constraints, or parent-chain limitations, a client may instead compare tips, finalized checkpoints, and anchor sets from multiple independently selected peers. This multi-peer mode can reduce reliance on a single node, but it is a weaker trust model and does not provide the same guarantee as authenticated parent-chain verification.

Thus, the client model balances minimal local storage with verifiability. Clients depend on nodes for availability, but

not for integrity or completeness of returned extension-block contents, provided that parent-chain anchors are verified.

K. EVALUATION METRICS

The experimental evaluation focuses on metrics directly related to distributed retrieval efficiency in the prototype setting.

- **End-to-end query latency**: the elapsed wall-clock time between issuing a client query and receiving the reconstructed response.
- **Number of sub-queries**: the number of intervals produced by the greedy decomposition algorithm for a given client request.
- **Per-subquery latency**: the execution time of each individual sub-query, measured separately by the dispatcher.
- **Node participation**: the set of nodes contacted for each query, including both successful and attempted nodes.
- **Returned data volume**: the total number of extension blocks retrieved, also reported as the total number of transferred bytes.
- **Coordination overhead**: the difference between the end-to-end query latency and the latency of the slowest successful sub-query, reflecting orchestration and aggregation costs.

Taken together, these metrics enable the analysis of both the algorithmic behavior of the decomposition mechanism and the systems-level performance of distributed retrieval. In particular, the number of sub-queries captures the extent to which a request interacts with the retention topology, while node participation and coordination overhead reveal the execution cost introduced by parallel retrieval and failover.

L. IMPLEMENTATION-SPECIFIC MEASUREMENT MODEL

The implementation follows a coordinator-based execution model. For each client request $[s, e]$, the coordinator first invokes the greedy retention-aware splitter in order to derive a sequence of interval assignments. Each assignment is then submitted to the asynchronous dispatcher together with its corresponding candidate node set. The dispatcher executes all interval assignments concurrently. Within an individual interval assignment, candidate nodes are attempted sequentially in randomized order until one node succeeds. Consequently, the protocol exhibits two complementary behaviors: parallelism across interval assignments and failover within a single assignment. Because sub-queries are launched concurrently, the overall query latency is primarily determined by the slowest successful interval retrieval together with the coordination overhead. Therefore, a larger number of sub-queries does not necessarily imply proportionally higher latency. Instead, the observed latency depends on how evenly the work is distributed and on whether the queried intervals activate slow archive nodes or repeated failovers.

IV. RESULTS AND ANALYSIS

This section evaluates the retrieval path of RAVEN on a 24-node prototype deployment consisting of regular nodes with heterogeneous retention windows and archive-capable nodes with complete extension-block history. Four retrieval strategies are compared. *Archive-only* sends each request to an archive node. *Naive whole-range* retrieves the entire requested interval from a single eligible node without decomposition. *RAVEN sequential* first applies retention-aware decomposition and then retrieves the resulting sub-queries sequentially. *RAVEN parallel* applies the same decomposition but dispatches the sub-queries concurrently.

The results reported in this section focus on the full-history retrieval workload. Each request spans the finalized extension-block history and therefore crosses all retention regions in the deployed topology. This workload is not intended to model the most common interactive message lookup. Instead, it evaluates a synchronization-style worst case in which both the benefit and the cost of retention-aware decomposition are exposed. Each strategy is evaluated at concurrency levels $c \in \{1, 4, 8, 16\}$, with 90 measured requests per strategy and concurrency level. Unless stated otherwise, latency values are reported as medians and tail latency refers to the 95th percentile. The results should be interpreted as prototype-scale evidence for retrieval behavior, not as a complete scalability, adversarial robustness, or fault-tolerance claim.

A. RETENTION-AWARE DECOMPOSITION AND ARCHIVE LOAD

The full-history workload activates the retention-aware splitter because the requested interval crosses the lower retention bounds of the regular-node tiers. For successful full-history requests, the baseline strategies issue a single retrieval request. In contrast, the RAVEN variants produce four topology-aligned sub-queries: an archive prefix and three regular-node suffix regions. In the evaluated topology, the resulting intervals are $[1, 8000]$, $[8001, 15000]$, $[15001, 18500]$, and $[18501, 19999]$. Thus, the number of sub-queries is determined by the number of retention-boundary transitions, not by the absolute number of requested blocks.

The primary effect of this decomposition is visible in the serving-load distribution. Figure 2 reports the fraction of returned bytes served by archive-capable nodes. Archive-only and Naive whole-range retrieve the complete interval from archive-capable nodes, so archive nodes serve 100% of the returned bytes. In contrast, both RAVEN variants reduce the archive byte share to 39.8%, with the remaining 60.2% served by regular nodes that cover the corresponding suffix intervals. This confirms that the splitter routes retrieval according to the retention topology and prevents full-history retrieval from being served entirely by archive nodes.

Figure 3 shows the same behavior at the level of individual nodes. Archive-capable nodes remain among the highest-load nodes because only they cover the historical prefix. However, regular nodes serve a substantial fraction of the total returned data. RAVEN therefore does not remove the need for archive nodes; rather, it confines archive use to the range that cannot be served by regular nodes and distributes the remaining suffix retrieval across the regular-node tiers.

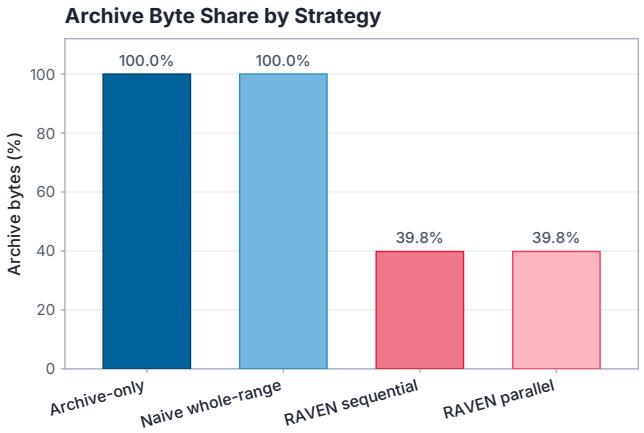


FIGURE 2. Share of returned bytes served by archive-capable nodes. In the full-history workload, the two baseline strategies obtain the full interval from archive-capable nodes, whereas RAVEN confines archive use to the historical prefix and serves 60.2% of the returned bytes from regular nodes.

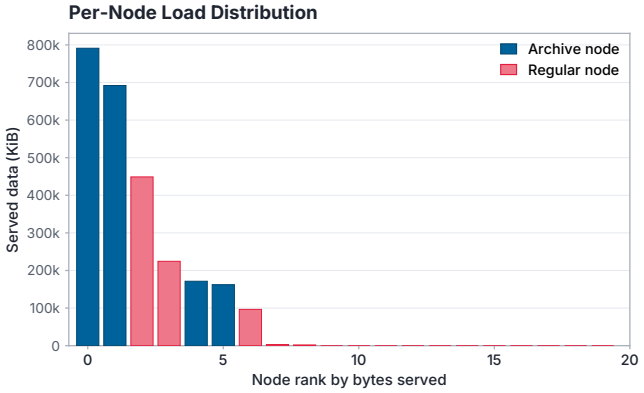


FIGURE 3. Per-node serving load ranked by bytes served and colored by node role. Archive-capable nodes serve the historical prefix, while regular nodes absorb the suffix intervals covered by their retention windows.

B. COMPLETION LATENCY

Figure 4 reports median end-to-end completion latency as a function of client concurrency. At $c = 1$, RAVEN parallel completes a full-history request in 835 ms, compared with 1073 ms for Archive-only, 1127 ms for Naive whole-range, and 1177 ms for RAVEN sequential. This corresponds to speedups of 1.28, 1.35, and 1.41, respectively. The result shows that, when contention is low, the latency reduction obtained by retrieving the retention regions concurrently exceeds the overhead of splitting, dispatch, validation, and merge.

The advantage decreases as offered concurrency increases. At $c = 16$, RAVEN parallel no longer has the lowest median completion latency: Archive-only and Naive whole-range complete in 10.9 s and 11.2 s, respectively, while RAVEN parallel completes in 12.1 s. This behavior is consistent with the expected cost of fan-out under contention. A RAVEN request issues multiple sub-queries and therefore creates more node-level work than a single whole-range request. Under heavier load, queuing and retries can offset the benefit of parallel interval retrieval.

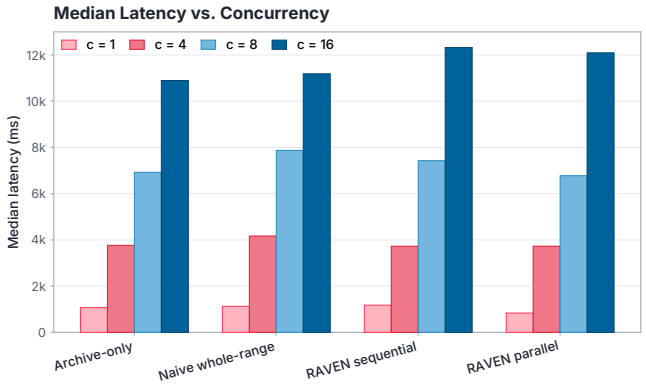


FIGURE 4. Median end-to-end completion latency as a function of concurrency. RAVEN parallel is fastest at low concurrency, while the advantage decreases under heavier load as fan-out and queuing costs become more visible.

These results support a bounded latency claim. RAVEN parallel improves full-history completion latency in the low-to-moderate concurrency regime, but it is not a uniformly faster replacement for single-node archive retrieval under all load levels.

C. PROGRESSIVE AVAILABILITY

Completion latency measures the time until the entire requested interval has been reconstructed and validated. For history synchronization, however, a client may also benefit from partial availability: the time at which a subset of the requested history can first be processed. The two baseline strategies are all-or-nothing in this respect. Because they retrieve the full interval as one response, no block is available until the whole response completes.

RAVEN exposes intermediate results because each retention-aligned sub-query can complete independently. Figure 5 reports the median time to the first returned sub-result, to 50% and 90% of the requested blocks, and to full completion at $c = 16$. Archive-only and Naive whole-range expose no data until approximately 10.4 s and 10.7 s, respectively. RAVEN parallel returns its first validated sub-result after 1.06 s, while RAVEN sequential returns its first sub-result after 3.64 s. Thus, the improvement is due both to decomposition and to parallel dispatch. At lower concurrency the same ordering is observed; for example, at $c = 1$, the first RAVEN-parallel sub-result is available after 46 ms.

This distinction is important because RAVEN's main benefit is not only final completion time. In workloads where a client can process or display retrieved history incrementally, retention-aware parallel retrieval substantially reduces the time to first usable data.

D. LOAD, RETRIES, AND SATURATION

At the highest evaluated concurrency, completion probability differentiates the strategies more clearly than median latency. At $c = 16$, RAVEN parallel completes 92.2% of requests, compared with 87.8% for Archive-only, 82.2% for Naive

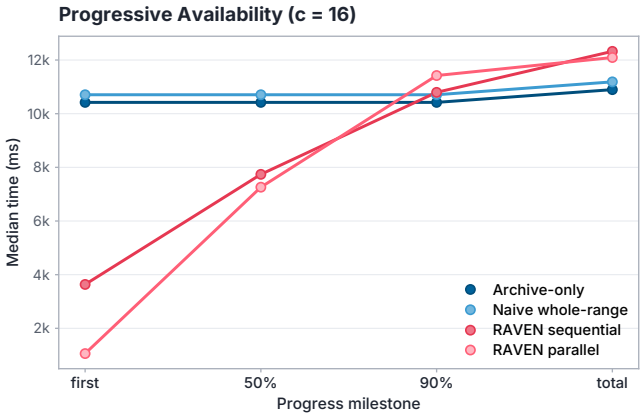


FIGURE 5. **Progressive availability at $c = 16$. The baselines expose no data until the full response completes, whereas RAVEN parallel returns an initial validated sub-result after 1.06 s and reaches later milestones as the remaining sub-queries complete.**

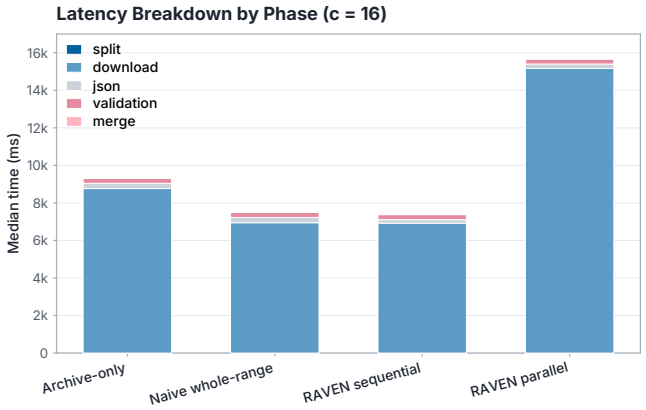


FIGURE 6. **Median latency decomposed by phase at $c = 16$. The retrieval/response phase dominates completion time, while splitting, client-side JSON processing, validation, and merge are comparatively small in the measured prototype.**

whole-range, and 66.7% for RAVEN sequential. The sequential RAVEN variant performs poorly under stress because each sub-query lies on the critical path. A timeout or slow response in any segment delays the entire request.

Parallel dispatch shortens this critical path and allows independent segments to fail over concurrently. The improvement is not free: RAVEN parallel records a higher retry rate than the baselines, indicating that the dispatcher contacts more candidate nodes before obtaining a complete response. This is the expected tradeoff of timeout-bounded fan-out. It improves completion robustness in the evaluated stress condition, but it also increases node-level request pressure.

The timing breakdown in Figure 6 further explains this behavior. The dominant component is the retrieval/response phase, which includes node-side response construction, network transfer, and receipt of the serialized response. Splitting, client-side JSON processing, validation, and merge contribute comparatively little in the measured prototype. Therefore, the main performance limit in this experiment is not the client-side splitter or verifier, but the cost of obtaining complete sub-query responses from serving nodes under load.

E. SUMMARY OF FINDINGS

The evaluation supports three conclusions. First, the retention-aware splitter performs the intended topological routing: in the full-history workload, RAVEN confines archive-node use to the historical prefix and reduces the archive byte share from 100% to 39.8%. Second, parallel dispatch improves the latency of full-history retrieval in the low-to-moderate concurrency regime and provides substantially earlier partial availability than the single-request baselines. Third, these benefits are load-dependent. At the highest evaluated concurrency, RAVEN parallel has the highest completion rate but not the lowest median completion latency, because fan-out increases node-level pressure and retry activity.

These findings define the contribution more precisely. RAVEN should not be interpreted as a universally lower-latency retrieval strategy. Its benefit appears when a request crosses retention regions or when early partial availability and reduced archive load are important. For small or single-region requests, the additional coordination overhead may outweigh the benefit of decomposition. The prototype therefore demonstrates the feasibility and tradeoffs of retention-aware verifiable retrieval, rather than establishing general scalability, adversarial robustness, or production-wide performance.

V. CONCLUSION

This paper presented RAVEN, a retention-aware retrieval architecture for encrypted message data stored in off-chain extension blocks and anchored by parent-chain commitments. In the proposed model, each protocol-participating parent-chain height is associated with an extension identifier, which allows clients to verify retrieved extension blocks against finalized parent-chain records without storing full message data on-chain.

The main technical contribution is a retention-aware query mechanism that partitions a client request according to node coverage boundaries and dispatches the resulting sub-queries to candidate nodes. This design exploits heterogeneous retention: archive-capable nodes provide historical coverage, while regular nodes serve the recent suffixes retained under their local storage policies. The prototype evaluation on a 24-node containerized deployment shows that, in the full-history workload, RAVEN reduces the archive byte share from 100% to 39.8% by shifting 60.2% of the returned data to regular nodes. The parallel variant also improves completion latency in the low-to-moderate concurrency regime and provides substantially earlier partial availability than the single-request baselines.

The assessment also identifies the limits of the approach. RAVEN should not be interpreted as a universally lower-latency retrieval strategy. Under heavier contention, fan-out increases node-level request pressure and retry activity, and the median completion-latency advantage over single-node

retrieval is not retained. The main demonstrated benefits are therefore more precise: reduced archive-node dependence, topology-aware load distribution, earlier availability of partial results, and improved completion behavior under the evaluated stress condition.

The present study remains limited to retrieval behavior in a controlled prototype setting. It does not validate full end-to-end messaging semantics, wide-area deployment behavior, write-path spam economics, parent-chain light-client verification costs, explicit reorganization handling under live network conditions, adversarial retrieval behavior, storage incentives, or metadata privacy. The results should therefore be interpreted as evidence for the feasibility and tradeoffs of retention-aware verifiable retrieval, rather than as a complete validation of a scalable, fault-tolerant, or privacy-preserving messaging system.

Future work should evaluate RAVEN in larger and geographically distributed deployments, study multi-client contention and load-aware node selection, measure behavior under node failures and adversarial retention claims, and integrate explicit parent-chain light-client verification. Further work is also needed on write-admission policies, storage incentives, privacy-preserving retrieval, and production-grade mechanisms for handling reorganization events and long-term archive availability. Finally, exploring how existing cryptographic protocols that guarantee forward secrecy can be used to strengthen user privacy.

VI. ACKNOWLEDGMENT

ChatGPT-5.5 was used for proofreading.

REFERENCES

[1] "How to decentralize the internet: A focus on data," *Computer Networks*, vol. 236, p. 109947, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128623003560

[2] "Blockchain-based solutions for mobile crowdsensing: State of the art, challenges and future directions," *Current Opinion in Systems and Software*, vol. 4, p. 100589, 2023. [Online]. Available: https://doi.org/10.1016/j.cosrev.2023.100589

[3] A. Punia, P. Gulia, N. S. Gill, E. Ibeke, C. Iwendi, and P. K. Shukla, "A systematic review on blockchain-based access control systems in cloud environment," *Journal of Cloud Computing*, vol. 13, no. 1, p. 146, 2024. [Online]. Available: https://doi.org/10.1186/s13677-024-00697-7

[4] "A privacy-aware authentication and usage-controlled data access framework," *Computers & Security*, vol. 139, p. 103728, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404824003559

[5] C. S. Leberknight, M. Chiang, and H. V. Poor, "A taxonomy of internet censorship and anti-censorship," *IEEE Communications Surveys & Tutorials*, vol. 13, no. 4, pp. 1–23, 2010.

[6] M. Shirali, T. Tefke, R. C. Staudemeyer, and H. C. Pöhls, "A survey on anonymous communication systems with a focus on dining cryptographers networks," *IEEE Access*, vol. 11, pp. 146 531–146 562, 2023.

[7] Y. He, M. Zhang, X. Yang, J. Luo, and Y. Chen, "A survey of privacy protection and network security in user on-demand anonymous communication," *IEEE Access*, vol. 8, pp. 67 345–67 361, 2020.

[8] B. Nelson, C. F. Griggio, Z. Sramek, and A. Askarov, "Metadata privacy beyond tunneling for instant messaging," in *IEEE European Symposium on Security and Privacy Workshops (EuroS&P Workshops)*. IEEE, 2024, pp. 123–134.

[9] J. Machado, "Exif metadata – privacy and security," Jul. 2022.

[10] X. Xu, Z. M. Mao, and J. A. Halderman, "Internet censorship in china: Where does the filtering occur?" in *Proc. Int. Conf. Passive and Active Network Measurement (PAM)*, 2011, pp. 133–142.

[11] Z. Rosson, F. Anthonio, C. Tackett, and A. N. team, "Shrinking democracy, growing violence: Internet shutdowns in 2023," Access Now #KeepItOn Coalition, Annual Report, May 2024, 283 documented internet shutdowns across 39 countries—the highest number on record. [Online]. Available: https://www.accessnow.org/wp-content/uploads/2024/05/2023-KIO-Report.pdf

[12] M. Marlinspike and T. Perrin, "The signal protocol," Open Whisper Systems, 2014, [Online]. Available: https://signal.org/docs/.

[13] O. Project, "Session: A private messenger built on blockchain," 2020, [Online]. Available: https://getsession.org/.

[14] M. Hodgson and A. L. Pape, "Matrix: An open network for secure, decentralized communication," in *Proc. Real-Time Communication Conf.*, 2019, [Online]. Available: https://matrix.org.

[15] Status Research & Development GmbH, "The Status network: A strategy towards mass adoption of Ethereum," White paper, 2017, [Online]. Available: https://status.app/whitepaper.pdf.

[16] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008, available: https://bitcoin.org/bitcoin.pdf.

[17] D. Chaum, "Untraceable electronic mail, return addresses, and digital pseudonyms," *Communications of the ACM*, vol. 24, no. 2, pp. 84–90, 1981.

[18] D. Goldschlag, M. Reed, and P. Syverson, "Onion routing for anonymous communication," *Communications of the ACM*, vol. 42, no. 2, pp. 39–41, 1999.

[19] P. Saint-Andre, "Extensible messaging and presence protocol (xmpp): Core," IETF RFC 6120, Mar. 2011, available: https://www.rfc-editor.org/rfc/rfc6120.

[20] C. Grothoff, "Gnunet — secure peer-to-peer networking," in *Proc. Int. Workshop Peer-to-Peer Systems (IPTPS)*, 2003.

[21] M. Abdulaziz, D. Çulha, and A. Yazici, "A decentralized application for secure messaging in a trustless environment," in *Proc. Int. Congr. Big Data, Deep Learning and Fighting Cyber Terrorism (IBIGDELFT)*, Ankara, Turkey, 2018, pp. 1–7.

[22] C. E. C. González and F. J. C. Romero, "Security issues of a decentralized blockchain-based messaging system," in *Proc. Congreso Int. Innovación y Tendencias en Ingeniería (CONIITI)*, 2021, pp. 1–7.

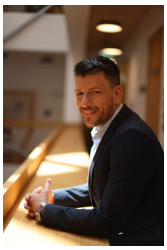
[23] A. Kuchimanchi, M. Vagdevi, M. Reddy, G. Avugaddi, and S. S. Kumar, "Chatease: A blockchain based chat application," in *Proc. 2nd Int. Conf. Appl. Artif. Intell. Comput. (ICAAIC)*, 2023, pp. 1171–1176.

[24] S. K. B. V., B. S. V., H. C. M., A. K., and H. H. K., "A blockchain-enabled chat application system for secure communication: Design and implementation," in *Proc. Int. Conf. Electron. Renewable Syst. (ICEARS)*, 2025, pp. 944–949.

[25] M. J. Baucas and P. Spachos, "Secure private blockchain-based instant messaging platform for social media services," *IEEE Network Letters*, vol. 6, no. 3, pp. 1–5, Jun. 2024.

[26] R. Saini, N. Saini, A. Aleem, and A. Singla, "P2p communication system using blockchain," in *Proc. 3rd Int. Conf. Commun., Security, Artif. Intell. (ICCSAI)*, 2025, pp. 1262–1268.

[27] A. Greubel, S. Pohl, and S. Kounev, "Quantifying measurement quality and load distribution in tor," in *Proceedings of the 36th Annual Computer Security Applications Conference*, ser. ACSAC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 129–140. [Online]. Available: https://doi.org/10.1145/3427228.3427238



ALEKSANDAR TOŠIĆ received the B.Sc., M.Sc., and Ph.D. degrees in computer science from the Faculty of Mathematics, Natural Sciences and Information Technologies, University of Primorska, Koper, Slovenia, in 2011, 2016, and 2022, respectively. His doctoral research addressed tradeoffs in using blockchain technology for security, privacy, and decentralization.

He is currently a Scientific Associate with InnoRenew CoE, Institute Andrej Marušič, University of Primorska, Izola, Slovenia, and an Assistant Professor with the Department of Information Sciences and Technologies, Faculty of Mathematics, Natural Sciences and Information Technologies, University of Primorska, Koper, Slovenia. He has coauthored more than 20 peer-reviewed scientific papers and his research interests include distributed computing, blockchain protocols, edge computing, artificial intelligence infrastructure, sensor networks, optimization, algorithms, and data structures.

Dr. Tošić received the Award for Scientific Excellence of the University of Primorska in 2026, the Solemn Charter of the University of Primorska in 2023, and the University Recognition for Academic and Research Achievements in 2021. He is a member of the Government Council for Digital Transformation of the Republic of Slovenia and has contributed to the research community through student supervision, project leadership and editorial work.

EKATERINA BOCHVAROSKA received the B.Sc. degree in computer science from the Faculty of Mathematics, Natural Sciences and Information Technologies, University of Primorska, Koper, Slovenia, in 2025, where she is currently pursuing the M.Sc. degree in computer science.

She is currently working as a developer in the Port of Koper, Slovenia, where she is involved in software development, data analysis, and digital solutions for logistics and port operations. Her research interests include decentralized systems, distributed computing, blockchain technologies, computer networks, algorithms, data structures.