Project seminar: Decentralised Message Protocol Built on Top of Blockchain

Mentor: Aleksandar Tošić

Student: Ekaterina Bochvaroska 89252018

Faculty of Mathematics, Natural Sciences and Information Technologies, 2026

Project Goal

The goal of this project is to design and implement a decentralized messaging protocol that:

- removes dependence on centralized servers
- avoids storing full message payloads directly on-chain
- allows efficient retrieval of message history
- supports networks where nodes have different storage capacities

Challanges

A decentralized messenger faces several engineering challenges:

- if messages are stored on-chain, the system becomes heavy and slow
- if everything is stored off-chain without verification, trust is reduced
- if all nodes store all data, storage requirements become unrealistic
- if nodes store only partial history, retrieval becomes more difficult

Design

The design is based on four principles:

1. **Minimal on-chain footprint**

Only compact references should be kept on-chain.

2. **Off-chain message storage**

Actual encrypted message data should be separated from the blockchain.

3. **Heterogeneous participation**

Nodes should be allowed to store different amounts of history.

4. **Retrieval-aware design**

The protocol must explicitly support efficient distributed queries.

High-Level Architecture

The system uses a dual-layer architecture:

Blockchain layer

- provides ordering
- provides cryptographic anchoring
- stores references to extension data

Extension layer

- stores actual encrypted messages
- pruned according to retention policy
- is queried directly by clients

Client layer

- discovers node capabilities
- issues interval queries
- reconstructs returned results

Extension Block Model

Each blockchain block is associated with an extension block containing the actual data. Where:

- h = block height
- parent Block Hash = link to corresponding blockchain history
- T = transactions or encrypted message records
- hash = commitment of the extension block

Node Types

The implementation distinguishes between two main node roles:

Regular nodes

- retain only a suffix of recent extension blocks
- lower storage cost
- suitable for ordinary participation

Archive nodes

- retain the complete extension-block history
- support long-range queries
- guarantee access to historical data

Retention Model

Each node maintains a retention window *Rn.*

At chain height h, a node can serve interval *[s,e]* if:

$$h - Rn + 1 \leq s \text{ and } h \geq e$$

This means:

- recent blocks are available from many nodes
- older blocks become concentrated in archive nodes
- storage burden is distributed unevenly but predictably

Query Problem

A client wants to retrieve a block interval *[s,e]*.

However:

- one node may not have the full interval
- data may be distributed across multiple retention windows
- archive nodes should not be contacted unnecessarily
- failed nodes must not break the query

The retrieval process must be distributed, efficient, and fault-tolerant.

Query Decomposition Idea

The protocol does not use fixed-size chunking.

Instead, it uses retention-aware decomposition:

- examine which nodes can serve which parts of the interval
- split the query only when the candidate node set changes
- keep each segment as large as possible
- preserve fallback candidates for each segment

Greedy Splitting Algorithm

The query decomposition works as follows:

1. start from the right endpoint of the query
2. find the first node whose retention lower bound covers that endpoint
3. create the largest interval that can be served by that candidate set
4. move leftward and repeat
5. merge adjacent intervals with the same candidate set

Query Dispatcher

For each sub-query:

- candidate nodes are shuffled
- one node is randomly chosen from the candidate set
- if it fails, the next candidate is tried

Across sub-queries:

- all interval requests are launched asynchronously

Execution model

- parallelism across intervals
- failover within intervals

Client Design

Clients are intentionally lightweight.

A client:

- obtains current node metadata
- learns retention capabilities
- submits an interval query
- receives returned extension blocks
- verifies them against chain commitments
- reconstructs the requested sequence

Development Plan (Weeks 1–4)

Week 1 — System Design

- finalize architecture (blockchain + extension layer)
- define data structures and interfaces

Week 2 — Storage & Nodes

- implement block, extension storage and nodes
- add retention pruning

Week 3 — Message Pipeline

- implement message encryption
- store messages in extension blocks

Week 4 — Query Decomposition

- implement greedy interval splitting
- test retention-aware partitioning

Development Plan (Weeks 5–8)

Week 5 — Query Execution

- implement dispatcher
- add parallel execution and retries

Week 6 — System Integration

- connect all components end-to-end
- test full query lifecycle

Week 7-8 — Evaluation & Finalization

- run experiments (workload + metrics)
- prepare documentation
- prepare results and presentation

Thank You!

Questions