



Waku Message UID

🕒 Created time	@January 16, 2023 12:08 PM
🕒 Last edited time	@January 17, 2023 6:40 PM
📏 Scope	Product
☰ Tags	
🔗 Issue link	
➤ Related pages	

Context and previous attempts

Message deduplication in Waku Relay (Gossipsub)

The *Message Cache* and the *Seen Cache* perform message deduplication in Waku Relay. These two deduplication structures rely on a unique ID at the moment of writing, which is computed as follows:

```
MessageId = HASH(WakuMessageBuffer)
```

These structures provide a limited-time message deduplication capability to keep the memory footprint low. The *Message Cache* provides a short-term deduplication capability (~5 heartbeat periods). The *Seen Cache*, implemented as a bloom filter, provides a longer-term deduplication capability (~2 minutes).

Based on the fact that `WakuMessage` is formatted using protocol buffers, and this serialization mechanism is not deterministic, there is a possibility that the *Seen Cache* fails to lead to duplicated messages by just reordering the different fields that are serialized.

Records deduplication in Waku Archive

At the moment of writing, the Waku Archive SQLite backend implementation considers two messages equal if both messages share the same `pubsub_topic`, `content_topic`, `timestamp` and `payload`. This criterion is baked into the SQLite's table `PRIMARY KEY`. Messages that do not match the criteria are treated as duplicates.

At the moment of querying via the Waku Store protocol, the results are ordered by the columns `stored_at` (equal to the message timestamp if present or to message arrival time otherwise) and `id` (computed as `Sha256(content_topic, payload)`).

Records retrieval via Waku Store

At the moment of writing, the *Waku Store* protocol provides an API supporting retrieving a filtered list of messages from a remote node. Additionally, as the maximum size supported for a message is 1MB, the *Waku Store* query protocol supports a pagination mechanism. This helps reduce the amount of data downloaded per query to a maximum of 100 messages (100 MB).

However, there is a limitation when verifying missing messages in a node's history. Downloading large amounts of data, in the order of hundreds of megabytes or even gigabytes, is not a time and bandwidth-efficient method for this task.

Message delivery order guarantees in Waku Relay (Gossipsub)

Gossipsub protocol, due to its flood-distribution nature, cannot guarantee that all messages will follow the same distribution path. So there is no ordering guarantee per se.

But if two messages are published separately enough in time, we can assume that both messages will be delivered in a predictable order. The maximum latency of the network determines this minimum separation.

Goal

Definition of a Waku message uniqueness identifier that can be used to deduplicate Waku messages across the solution.

No goal

Build a blockchain. Consistency guarantees between archive-capable nodes are out of the scope of the present document.

Use cases

- Message deduplication in the network.
- Message deduplication in the Waku Archive backend (e.g., in a shared backend setup).
- Bandwidth-efficient Waku Archive synchronization.

Requirements

- Length limited.
- Low to negligible collision probability.
- Not leaking information (e.g., sender's key).
- Application specific (e.g., lexicographic sortable).
- Uniqueness is global to the network.
 - Global to all the nodes publishing in a certain network (pub-sub topic).
 - Global to the Gossipsub's pub-sub topics certain store node is subscribed to.

Proposal: *Message UID*

- Add a **mandatory** attribute of arbitrary application-specific content limited to 32 bytes (2^{256} possibilities) called `WakuMessage.uid`. Mandatory if archiving capabilities are required.
- Deprecate `WakuMessage.timestamp` field.

Application specific

Applications should specify the schema for the identifier attribute. The UID schema will affect the privacy, security and message collision probability of the application's messages:

- **Timestamp (e.g., int64 Unix Epoch nanoseconds timestamp):**
 - **PRO:** Simple, performant generation, “backwards compatible”, fine grain sortable at archive query results (precision in ns).
 - **CON:** Prone to collision/message duplication, traceable/graph learning

- **ULID:**
 - **PRO:** Medium complexity, performant generation, negligible collision probability, fine grain sortable at archive query results (precision in ms).
 - **CON:** traceable/graph learning
- **UUID (e.g., UUID v4):**
 - **PRO:** Medium complexity, performant generation, negligible collision probability, no traceability (random data).
 - **CON:** Coarse sortability at archive query time.
- **Application-specific schema (e.g., Sha256 signature, Encrypted meta-info):**
 - **PRO:** Negligible collision probability (if the content is well thought), contains metadata, non-traceable (looks like random data).
 - **CON:** High complexity, not-so-performant generation (hashing, encryption), not sortable at archive query time.

Waku Relay: message deduplication

Based on the low collision probability of some of the schemas described above, this UID attribute could be used as the cache ID, replacing the need to hash the entire byte array by just deserializing the `uid` attribute.

A message that reuses the same ID with a different payload within the *Message Cache* window won't be relayed. In the same way, if it is replayed within the *Seen Cache* window, it won't be received by subscribers.

Waku Archive and Waku Store: durable streams



The terms *event*, *message*, and *record* are synonyms in the *Waku Archive* context. Thus they can be used interchangeably. Although the *record* term is preferred over the others when talking about individual items in an events log.

From the Waku product document's terms and concepts:

Waku platform events are organized and durably stored in **topics**, also called **content topics**. And a sequence of events published on the same **topic** constitutes an **event stream**.

Event recording

The *Waku Archive* stream durability service is responsible for recording and persisting in a long-term storage system the events that occurred in a certain event stream. This persistence follows two rules:

- Messages should be stored following the arrival order (FIFO).
- Messages should be deduplicated based on the UID attribute (the same criteria used in the *Waku Relay* layer).

Optionally, messages can be tagged with arrival time timestamps for coarse-ordering purposes.

Queryable event log

Additionally to the event recording system, the *Waku Archive* service features a searchable log functionality and provides an interface for retrieving messages from the previously described event recording system. And the *Waku Store* historical messages query protocol sits on top of this interface, making it easily accessible through a remote procedure call (RPC) interface.

Waku Store's bandwidth-consumption optimization

Based on the assumption that UID identifiers are globally unique to all messages. We can understand archive-capable nodes as key-value stores. The UID would be the key, and the message would be the value.

With that approach, the current *Waku Store* protocol RPCs can be extended to support a message ID query mechanism. The new *Waku Store* protocol APIs will look like this:

- Query a list of messages based on certain filter criteria (e.g., network, content topic, time range, etc.).
- Query a list of message UIDs based on certain filter criteria (e.g., network, content topic, time range, etc.).
- Query messages by a UIDs list.

With a maximum size of 32 bytes, the number of UIDs per query response can be higher, saving bandwidth and reducing the “time-to-sync” metric.

It is discretionary to the application in which APIs are used and when. For example, bootstrapping the node’s history and getting the first 50 messages that fit the screen for a hypothetical messaging app can make a difference in the perceived UX. Once bootstrapped the node’s message history, the same application could use the UID-based query mechanism to efficiently retrieve the missing messages and complete the rest of the messages’ history.

Conclusions and future work

This proposal extends the current model and tries to unify the Waku platform’s relay (*Waku Relay*) layer with the platform’s stream durability functionality (*Waku Archive*) and the stream history query functionality (*Waku Store*).

Eventually, this UID-based history synchronization mechanism has the potential to be evolved into a fully-fledged history synchronization mechanism. Due to this unified approach, it has the potential to be added as a Gossipsub extension adding a “durable stream capability” to the protocol.

A in depth privacy and security analysis is pending.