dm3 documentation

Release 1.0.0RC

Steffen Kux

Apr 04, 2023

SPECIFICATIONS

Contents:

CHAPTER

ONE

OVERVIEW

1.1 Abstract

*The **dm3** (**D**ecentralized **M**essaging for web**3** or **D**irect **M**essage **3**) protocol is an end-2-end **encrypted** peer-2-peer messaging protocol, rooted in **ENS** (Ethereum Name Service). It is **decentralized**, has no single-points-of-failure, has a **lean architecture** with minimum resource requirements, and enables **interoperability** with other messaging solutions while preserving the **self-sovereignty** of its users. The **dm3** protocol uses **ENS** as central registry for necessary contact information (like public keys, addresses of delivery services, . . .), stored in ENS text records, in combination with a standardized API to build an open ecosystem of message delivery services allowing to send messages from ENS name to ENS name.*

1.2 Motivation

Messaging (such as instant messages, chats, email, etc.) has become an integral part of most people's lives. Mobile devices (such as smartphones, tablets, laptops, etc.) with instant access to the Internet make it possible to be in touch with family, friends, as well as work colleagues, and customers at any time.

While email services are still largely decentralized and interoperable, the lack of appropriate spam protection methods other than blocking and censoring has resulted in only a few large providers interacting with each other, not to mention the fact that even today a large portion of email communication is mostly unencrypted.

Messaging services on the web2 (like WhatsApp, Signal, Telegram, . . .) have become closed silos, making cross-service or cross-app communication almost impossible. Although they mostly offer end-to-end encryption, some services may still have backdoors via the central service providers or can stop end-2-end encryption without the user's approval.

In the past months, several different approaches and tools have been presented in web3. Methods from the web3 such as key-based identification, encryption, and the availability of blockchain-based registries are being used. Many applications are built to meet user preferences, and several protocols have been presented. Trade-offs are often necessary - such as centralized services, application-related registries, or complex protocols. Interoperability across applications, services, and protocols is still limited, if possible at all.

With **dm3**, a protocol is presented, which is characterized by a very lean base protocol (DM3MTP - the **dm3** message transfer protocol), which can serve as a bridge between different services and can enable integration and interoperability with other services and different applications. The aim of **dm3** is to suggest a common base standard for web3 messaging, on which further protocols and applications can be built to create a silo-free, secure, self-determined, decentralized messaging ecosystem - based on web3 technology.

This allows users not only to have full control over their data and messages but also to choose the messaging app that best suits their needs and preferences, without the compromise of being limited to a particular ecosystem.

1.3 Base Architecture

The **dm3** protocol is designed as a lean peer-2-peer messaging protocol with consistent end-to-end encryption and sufficient decentralization through an open delivery service architecture.

Due to its simple base architecture, **dm3** is intended as a base protocol to bring together a variety of messaging applications and protocols so that true interoperability can be realized.

Required contact information such as public encryption keys and public keys to verify signatures as well as information on used delivery services are managed as text records in ENS (Ethereum Name Service) - the **dm3 profile**. This provides a general registry that can be accessed across applications and protocols. Thus, services using this standard do not have to rely on the technology and availability of one provider, nor does it result in the emergence of various incompatible silos in web3 as in web2.

1.4 Protocol Extensions

While the specification of the **Message Transport Protocol (DM3MTP)** focuses on a standardized format description for general messages and how to deliver those messages to a delivery service defined in the receiver's **dm3** profile, several optional protocol extensions are defined to cover further topics.

The **dm3** delivery service and **dm3** compatible app implementations MAY also use the following **dm3** protocol extensions:

- **Message Access Specification**: Specifies how received messages on a delivery service can be accessed.
- ***Message Storage Specification***: Specifies how messages are persisted after they are delivered.
- **Public Message Feed Specification**: Specifies how a public message feed is created and accessed.
- **Intra Delivery Service Messaging Specification**: Specifies additional features for messaging if sender and receiver are using the same delivery service.
- **Group Messaging Specification**: Specifies a protocol extension to enable group messaging.
- **Encryption and Signing Key Derivation Specification**: Specifies how to derive keys from the wallet key.
- **Privacy Onion Routing Specification**: Specifies a protocol extension to enable privacy-preserving onion routing.
- **Spam Protection Specification**: Specifies additional methods, based on web3 technology, that prevent spam on the receiver's side.
- **Layer-2 Registry Specification**: Specifies how to include a layer-2 or cross-chain registry into subdomains to extend the general registry
- **Top-Level Alias Specification**: Specifies how subdomains can be mapped to non-ENS-top-level domains. This is needed for instance for cross-chain interoperability.

The specifications for the protocol extensions are still in draft status and will be published soon.

CHAPTER TWO

MESSAGE TRANSPORT PROTOCOL (DM3MTP)

2.1 Registry

A central (but decentralized) registry is needed where a **dm3** compatible app, service, or protocol can lookup **dm3 profiles** of other users, containing

- Public keys,
- Links to delivery services.

The **dm3** protocol uses **ENS (Ethereum Name Service)** as a central (but decentralized) registry. The following text records are used for this purpose:

- `network.dm3.profile`: User profile entry
- `network.dm3.deliveryService`: Delivery service profile entry

The text records MUST be a URI containing the profile JSON string defined below.

The URI can be

- A data scheme (*data:...*) or
- A URL pointing to a profile JSON object (*HTTPS:... or IPFS:...*). To validate the integrity of the resolved profile JSON string, the URL MUST be a native IPFS URL or a URL containing a `dm3Hash` parameter containing the **SHA-256** hash of the JSON.

  Example `network.dm3.profile` text record entries:

  - `data:application/json,{profile...`
  - `https://delivery.dm3.network/profile/0xbcd6de065fd7...b3cc?dm3Hash=ab84f8...b50c8`
  - `ipfs://bafybeiemxf5abjwjz3e...vfyavhwq/`

The profiles can only be changed by creating a new profile JSON and changing the corresponding text record via an Ethereum transaction (if published on layer-1). Storing this information on layer-2 or linked via CCIP (Cross-Chain Interoperability Protocol) using subdomains, is possible, too. The specification thereof will be published in the protocol extension **Layer-2 Registry Specification**. This is currently under development and will be published soon.

Information read from ENS may be cached for performance reasons but the ENS TTL settings must be respected (to be fetched from the resolver).

2.1.1 User Profile

The user profile MUST contain:

- **Public Signing Key:** Key used to verify a message signature (ECDSA). The public signing key is the public key of a secp256k1 private/public key pair. How to generate or derive this key pair depends on the implementation of the client. The **Encryption and Signing Key Derivation Specification** proposes a method to derive those keys based on a signature of the wallet keys. The key is presented as base64-encoded string of the key's bytes (see key encoding).
- **Public Encryption Key:** Public key used to create the key (together with the private key of the sender) to encrypt a message. As default, the algorithm **x25519-chacha20-poly1305** is used (see *[NIR1]*). If needed (e.g., for compatibility reasons with an integrated protocol), a different encryption can be specified in the Profile Extension. Nevertheless, using the default encryption is highly recommended. The key is presented as base64-encoded string of the key's bytes (see key encoding).
- **Delivery Service List:** List with at least one delivery service' ENS name[1].

```
DEFINITION: UserProfile

{
  // Key used to encrypt messages
  publicEncryptionKey: string,
  // Key used to sign messages
  publicSigningKey: string,
  // ENS name list of the delivery services e.g., delivery.dm3.eth
  deliveryServices: string[],
}
```

Example UserProfile with fallback delivery service:

```
{
   "publicEncryptionKey":"nyDsUmYV4EDNCsG+pK...D=",
   "publicSigningKey":"MBpqhsSkxevwbYEGnXX9r...c=",
   "deliveryService": ["example_deliveryservice.eth","example_fallback-
↪deliveryservice.eth"]
}
```

Additional to the user profile, the user profile extension can be queried from the user's delivery service (see user profile extension). As this information may change and depend as well on the delivery service, it MUST be requested directly from the delivery service the message will be sent to.

Key encoding

Public keys published in the profiles are presented as base64-encoded strings of the key's bytes.

Example Key encoding

```
Key = [134, 57, 101, ..., 167]
KeyString = "jjllMO...qc="
```

[1] For information, on how to adapt **dm3** for ecosystems not based on Ethereum, see appendix Cross Chain Applications.

2.1.2 Delivery Service Profile

The delivery service profile MUST contain:

- **Public Signing Key:** Key used to verify a postmark signature (see **UserProfile**). The key is presented as base64-encoded string of the key's bytes (see key encoding).
- **Public Encryption Key:** Public key used to create the key (together with the private key of the sender) to encrypt a message. As default, the algorithm **x25519-chacha20-poly1305** is used. The key is presented as base64-encoded string of the key's bytes (see key encoding).
- **Delivery Service URL:** URL pointing to the delivery service instance.

As the encryption algorithm for the delivery service, the default algorithm **x25519-chacha20-poly1305** is mandatory.

```
DEFINITION: DeliveryServiceProfile

{
  // Key used to sign postmarks
  publicSigningKey: string,
  // Key used to encrypt delivery information
  publicEncryptionKey: string,
  // URL pointing to the delivery service instance
  url: string
}
```

Example: DeliveryServiceProfile

```
{
   "publicEncryptionKey":"nyDsUmYV4EDNCsG+pK...D=",
   "publicSigningKey":"MBpqhsSkxevwbYEGnXX9r...c=",
   "url": "https://example_deliveryservice"
}
```

2.2 Message Transport

Sending (and receiving) a message takes place in 3 steps, although only the first two steps are part of the **dm3 Message Transfer Protocol**.

1. The sender app **prepares and sends the message to the receiver's delivery service**. If the primary delivery service (first on the list) is not available, the next one from the list is contacted (and so on).
2. The **delivery service buffers and processes the message** (checks envelope, creates postmark to protocol time of delivery, optionally sends a notification to the receiver, . . .).
3. *The **message is picked up by the recipient**. As soon as the recipient reports the successful processing of the message to the delivery service, the latter deletes the buffered message. !!! This is not part of the "Message Transfer Protocol", as this depends on the implementation and objective of the delivery service. If the delivery service is following the **dm3 Access Specification** to serve **dm3** compatible clients, it offers a REST API to retrieve the messages, but a delivery service may also act as an interface/gateway to another protocol or application ecosystem, handling incoming messages according to its rules. !!!*

2.2.1 Workflow

Step 1: Preparation of the Message and Envelope

Get dm3 profile

1. Read the `network.dm3.profile` text record of the receiver's ENS name. If the profile record is not set, the message cannot be delivered. It has to stay with the sender until the potential receiver publishes his/her profile.
2. The content is specified as URI (Uniform Resource Identifier). The following types must be supported:
    1. **DATA:** The content is delivered as JSON. The data scheme MUST be `application/json`. Optionally, the JSON content is **base64** encoded. This must be specified as scheme extension `application/json;base64`. If not base64 encoded, the content might be URL-encoded.

       Example: `data:application/json,%7B%22profileRegistryEntry%22%3A...` `data:application/json;base64,eyJwdWJsaWNFbmNyeX...`

    2. **HTTPS:** The content is retrieved as a JSON object from a server and the `dm3Hash` URL parameter is used to check the integrity of the profile string.

       Example: `https://exampleserver/example?dm3Hash=0x12ab4...`

    3. **IPFS:** The content is retrieved as a JSON object using IPFS network.

       Example: `ipfs://QmU6n6n1Q...`

3. Interprete JSON object as **dm3 profile**.
4. Select the receiver's delivery service ENS name by reading the `deliveryServices` user profile entry at index `0`.
    1. Get the `network.dm3.deliveryService` text record of the delivery service's ENS name. The content is delivered as URI (data, HTTPS, or IPFS), as described above in point 2.
    2. Interprete JSON object as **dm3 delivery service profile**.
5. If the selected delivery service is unavailable, the sender MUST use the delivery service with the next higher index in the `deliveryServices` list as a fallback.

Create a Message and Envelope

1. Get ProfileExtension from delivery service
    - Read optional encryption parameters (like preferred encryption algorithm deviating from standard)
    - Read supported message types. Only supported messages must be sent.
2. Sign the message using the private sender signing key, using ECDSA.
3. Encrypt the message using the public encryption key of the receiver (part of the user profile). The default encryption algorithm is **x25519-chacha20-poly1305**. If a different algorithm is required (defined in the *ProfileExtension*), this should be used for encryption. If it is not supported by the sender, the default encryption is used.
4. Encrypt the delivery information using the public encryption key of the delivery service (part of the delivery service profile). The mandatory encryption algorithm is **x25519-chacha20-poly1305**.

```
sequenceDiagram
  participant AA as Alice' Client
  participant E as Registry (ENS)
  participant P as Profile Storage (e.g. IPFS)
  AA->>E: get network.dm3.profile for Bob's ENS name
```

(continues on next page)

(continued from previous page)

```
  E-->>AA: network.dm3.profile text record
  opt network.dm3.profile text record is an URL
    AA->>P: query Bob's profile
    P-->>AA: profileRegistryEntry
    AA->>AA: check profileRegistryEntry integrity
  end
  AA->>AA: sign message
  AA->>AA: encrypt message

  AA->>E: get network.dm3.deliveryService of Bob's delivery service
  E-->>AA: network.dm3.deliveryService text record
  opt network.dm3.deliveryService text record is an URL
    AA->>P: query delivery service profile
    P-->>AA: deliveryServiceRegistryEntry
    AA->>AA: check deliveryServiceRegistryEntry integrity
  end

  AA->>AA: sign envelope
  AA->>AA: encrypt deliveryInformation
```

Submit Message

1. Get ProfileExtension from delivery service
   - Read spam protection settings (see Profile Extension).
   - Check, if conditions are met. If not, the message must not be sent (as it will be discarded from the receiving delivery service anyway). The sender should be informed.
2. Submit the message to the delivery service using the URL defined in the delivery service profile.

Step 2: Message processing at the delivery service

1. Decrypt delivery information.
2. Apply filter rules from the receiver's profile extension. Discard the message if conditions are not met.
3. Create a postmark. The postmark protocols the reception and buffering of the message.
4. Buffer message. The delivery service is responsible to store the encrypted message until the receiver picks it up. A delivery service may decide to have a max holding time. It must be at least 1 month. If the receiver didn't fetch the message within this time, the message may be deleted. This time can be queried from the delivery service' properties
5. Optional: send notification(s) to the receiver that a message is waiting for delivery.

```
sequenceDiagram
  actor A as Alice
  participant AA as Alice' Client
  participant BD as Bobs's Delivery Service
  participant CC as Alice' Data Storage Service
  participant BB as Bob's Client
  A-->>AA: writes message
  AA->>BD: dm3_getProfileExtension
  BD-->>AA: retrieve Bob's "profileExtension"
```

(continues on next page)

(continued from previous page)

```
    AA->>AA: prepare message
    opt
      AA->>AA: prepare attachments
      opt
        AA-->>CC: store attachment data at service or IPFS
      end
    end
    AA->>BD: dm3_getDeliveryServiceProperties
    BD-->>AA: retrieve the delivery service' properties
    AA->>AA: prepare envelope
    AA->>BD: dm3_submitMessage
    BD->>BD: decrypt deliveryInformation
    BD->>BD: apply filter rules
    BD->>BD: add postmark
    BD->>BD: buffer message
    opt
       BD->>BB: notification
    end
```

2.2.2 Message Data Structure

The message data structure stores all data belonging to the message that can only be read by the receiver. The entire data structure is encrypted (based on the public key of the receiver).

The message data structure contains the following information:

- **Message:** *(OPTIONAL)* This string contains the actual message. For service messages (like READ_RECEIPT, RESEND_REQUEST, or DELETE_REQUEST), this field may be empty or undefined. The message MUST be **plain text** (UTF-8), optionally flavored with **Markdown** highlightings. If other encodings of the message are provided, those MUST be attached as attachment (embedded with data scheme only), still providing the text representation in the message string. Clients able to interpret the encoded attachment may display this instead of the original message string. Others will visualize the message text as plain text or Markdown formatting.
- **Metadata:** This object contains all meta information about the message. Some attributes are mandatory, others are optional. Also, application-specific attributes can be added. The MessageMetadata-Structure is described in detail below.
- **Attachments:** *(OPTIONAL)* Media or other files or special encodings of the message may be an attachment to a message, defined as an array of URIs (data, HTTPS, IPFS). Attachments are described in detail below.
- **Signature:** This is the signature with the sender's signature key on the SHA-256 hash of the message data structure without the signature field.

```
DEFINITION: Message Data Structure

{
   // message text
   // optional (not needed for messages of type READ_RECEIPT, DELETE_REQUEST, and RESEND_
↪REQUEST)
   message?: string,
   // metadata added to the message.
   metadata: MessageMetadata,
   // message attachments e.g. images as an array of URIs
```

(continues on next page)

(continued from previous page)

```
    // (optional)
    attachments?: string[],
    //the signature of the sender
    // sign( sha256( safe-stable-stringify( struct_without_sig ) ) )
    signature: string
}
```

2.2.3 Message Metadata Structure

The **message metadata structure** stores all meta information belonging to a message. While some attributes are mandatory, others are optional. If needed, application-specific meta information may be added, too.

The **message metadata structure** contains the following information:

- **To:** The ENS name the message is sent to.
- **From:** The ENS name of the sender
- **Timestamp:** The timestamp (unixtime in milliseconds) when the message was created.
- **Type:** Different types of messages can be sent. A **dm3** compatible messenger may not support all types in the UI but must at least handle not interpreted types meaningful (*example: the messenger doesn't support editing existing messages. It appends messages with the type* ***EDIT*** *as new messages at the bottom of the conversation*).
  - **NEW:** A new message.
  - **DELETE_REQUEST:** *(OPTIONAL)* This is a service message. The sender wants the referenced message deleted. The value **referenceMessageHash** points to the message to be deleted. If the receiver's messenger doesn't support the deletion of messages, it may ignore the message.
  - **EDIT:** *(OPTIONAL)* An already existing (old) message was edited (new version). The value **referenceMessageHash** points to the message to be replaced with a new version. If edit is not supported by the receiver's messenger, the message must be added as new.
  - **REPLY:** *(OPTIONAL)* This message is a direct reply to an existing message. The value **referenceMessageHash** points to the referenced message. If threads/references are not supported, it must be added as a new message.
  - **REACTION:** *(OPTIONAL)* This is a short referenced message, containing an emoji as a **message**, and the value **referenceMessageHash** points to the referenced message.
  - **READ_RECEIPT:** *(OPTIONAL)* This is a service message. The receiver sends this message back to the sender to signal that the message was received and displayed. Sending this message is optional and it may be ignored.
  - **RESEND_REQUEST:** *(OPTIONAL)* This is a service message. The value **referenceMessageHash** points to the referenced message. If possible (=available), the referenced message should be sent again.
- **Reference Message Hash:** *(OPTIONAL)* The hash of a previous message that the new one references. Must be set for message types (REPLY, DELETE_REQUEST, EDIT, REACTION, RESEND_REQUEST).
- **Reply Delivery Instruction:** *(OPTIONAL)* It is needed for compatibility reasons with other protocols/apps. The stored information MUST be delivered with any reply (e.g., a conversation or topic id, . . .) as meta information of the encryption envelope. It is neither evaluated nor altered from **dm3**.

```
DEFINITION: Message Metadata Structure

{
```

(continues on next page)

(continued from previous page)

```
    // receiver ens-name
    to: string,
    // sender ens-name
    from: string,
    // message creation timestamp
    timestamp: number,
    // specifies the message type
    type: "NEW" | "DELETE_REQUEST" | "EDIT" | "REPLY" | "REACTION" | "READ_RECEIPT" | "RESEND_REQUEST"
    // message hash of the reference message
    // optional (not needed for messages of type NEW)
    referenceMessageHash?: string,
    // instructions used by the receiver of the message on how to send a reply
    // optional (e.g., used for bridging messages to other protocols)
    replyDeliveryInstruction?: string,
    // any kind of additional metadata may be added.
    // This might be information needed by protocol extensions or app-specific meta information.
    ...
}
```

2.2.4 Attachments

Attachments can be any type of additional data or media files. These are organized as an array of URIs. Embedded content is encoded as data scheme, external data as URL or IPFS. A message can have no or an arbitrary number of attachments. The overall size of the message (inclusive of all embedded attachments) MUST be less than 20MB. The overall size of the message can be restricted additionally by the delivery service (see Delivery Service Properties.) If bigger media files need to be attached, the actual data need to be stored outside the message (still encrypted with the receiver's public key) and the attachment contains only the reference (URI with HTTPS or IPFS scheme). Otherwise, the attachment may be included with URI scheme data.

Different **dm3** compatible applications may handle attachments differently (visualization, embedding, or even ignoring them). Applications may optionally support other encodings than text/markdown for the message. These may be added as attachment and visualized instead of the original message text. It is the application's responsibility to do this properly.

Examples:

```
"attachments":["data:text/html;base64,dfEwwewGJsaWKklNyeX...",...],
```

```
"attachments":["data:image/jpeg;base64,dfEwwewGJsaWKklNyeX...",...],
```

```
"attachments":["https://exampleservice/exampleresource",...],
```

```
"attachments":["ipfs://AmE6mn1n64Q...",...],
```

2.2.5 Encryption Envelope Data Structure

The encryption envelope is the data structure that is sent to the delivery service. It contains delivery metadata and the encrypted message itself. The envelope is read and interpreted by the delivery service. However, the actual message is encrypted based on the receiver's key and signed with the sender's key so that it cannot be read or altered by the delivery service.

The encryption envelope contains the following data:

- **Message:** The encrypted message (Message Data Structure).
- **Metadata:** This object contains all meta information about the envelope. Some attributes are mandatory, others are optional. Also, application-specific attributes can be added. The EnvelopeMetadata Structure is described in detail below.
- **Postmark:** *(OPTIONAL)* A data struct with the information on the delivery status. Postmark is empty/undefined when the sender is sending the envelope to the delivery service. It is added by the delivery service and is encrypted based on the public key of the receiver.

```
DEFINITION: Encryption Envelope Structure

{
  // the message
  // encrypted based on the receiver's public encryption key
  message: string,
  // meta information for the envelope
  metadata: EnvelopeMetadata,
  // contains information added by the delivery service
  // encrypted based on receiver's public encryption key
  postmark?: string,
}
```

2.2.6 Envelope Metadata Structure

The **envelope metadata structure** stores all meta information belonging to an envelope. While some attributes are mandatory, others are optional. If needed, application-specific meta information may be added, too.

The **envelope metadata structure** contains the following data:

- **Version:** The protocol version of **dm3**.
- **Encryption Scheme:** The used encryption and signing algorithms. The default is **x25519-chacha20-poly1305**. If this field is not set (undefined), the default is being used.
- **Delivery Information:** A data struct with the delivery information needed by the delivery service (message metadata).

```
DEFINITION: Envelope Metadata Structure

{
  // dm3 protocol version (e.g., 1.0)
  version: string,
  // used encryption scheme of the message
  //optional: if not set, the default x25519-chacha20-poly1305 is taken
  encryptionScheme?: string,
  // datastruct with delivery info
```

(continues on next page)

(continued from previous page)

```
  // Delivery information object, encrypted based on the delivery servics' encryption key
  deliveryInformation: string,
  // any kind of additional metadata may be added.
  // This might be information needed by protocol extensions or app-specific meta information.
  ...
  // the signature of the sender
  // sign( sha256( safe-stable-stringify( struct_without_sig ) ) )
  signature: string,
}
```

2.2.7 Delivery Information

The delivery information contains all metadata needed by the delivery service to handle a message.

The data structure contains the following information:

- **To:** The ENS name the message is sent to.
- **From:** the ENS name of the sender
- **Delivery Instruction:** this is optional information. It is needed for compatibility reasons with other protocols/apps. The stored information (e.g., a conversation or topic id, . . .) will be delivered with any reply from the receiver. It is neither evaluated nor altered from **dm3**.

```
DEFINITION: Delivery Information

{
  // receiver ens-name
  to: string,
  // sender ens-name
  from: string,
  // instructions used by the delivery service on how to deliver the message
  // optional (used for bridging messages to other protocols)
  deliveryInstruction?: string
}
```

2.2.8 Postmark Data Structure

The postmark data structure contains information added by the delivery service regarding the delivery status.

It contains the following information:

- **Massage Hash:** the Hash (SHA-256) of the entire message.
- **Incoming Timestamp:** The unixtime in milliseconds when the delivery service received the message.
- **Delivery Information:** This is a copy of the delivery information provided in the envelope. As this info in the envelope is encrypted for the delivery service, it MUST be added from the delivery service to the postmark. The receiver can use this information to check if the sender of the message referenced in the Message Metadata) is the same as referenced in the envelope.
- **Signature:** the signature of the postmark from the delivery service. This is needed to validate the postmark information.

```
DEFINITION: Postmark

{
  // sha256( EncryptionEnvelope.message )
  messageHash: string,
  // timestamp of when the delivery service received the message
  incomingTimestamp: number,
  // a copy of the delivery information from the envelope the delivery service
  deliveryInformation: DeliveryInformation,
  // signature of the delivery service
  // sign( sha256( safe-stable-stringify( postmark_without_sig ) ) )
  signature: string,
}
```

2.3 API Delivery-Service (Incoming Messages)

For more detailed information about delivery services, see the appendix. Relevant to DM3MTP (protocol) is the API to deliver messages (encrypted envelopes) only.

The delivery service is a JSON-RPC service, following the JSON-RPC 2.0 specification (see also ***[RPC1]***).

To accept incoming messages, the delivery service MUST support the following JSON-RPC methods:

2.3.1 Submit Message

The submit message method is called to deliver the complete message envelope containing the delivery information and the encrypted message.

Methode

```
// call to submit a message
dm3_submitMessage
```

Request

The request delivers the encrypted envelope containing the message itself and the delivery information as EncryptionEnvelope.

```
// see description of EncryptionEnvelope data structure
EncryptionEnvelope
```

Response

The response is as defined in the JSON-RPC specification. In case of an error, an error message is returned.

Example

```
{
 "jsonrpc": "2.0",
 "error": {
   "code": -32600,
   "message": "Invalid Request"
 },
 "id": null
}
```

Error codes

For default JSON-RPC error codes see appendix.

Additinal, application specific error codes can be reported:

Error code	Error text	Description
-32000	Invalid input	Missing or invalid parameters.
-32001	Resource not found	Requested resource not found.
-32002	Resource unavailable	Requested resource not available.
-32003	Unautorized	The caller was not autorized to call this method.
-32004	Method not supported	Method is not implemented.
-32005	Limit exceeded	Request exceeds defined limit.
-32006	JSON-RPC version not supported	Version of JSON-RPC protocol is not supported.

If the message is rejected from the delivery service, the following error codes will be returned:

Error code	Error text	Description
-32010	Spam	The sender's address didn't fit to the requirered spam protection settings.
-32011	Too big	The size of the message exeeds the approved maximum size.

2.3.2 Get Properties of the Delivery Service

A **delivery service** provides a set of properties that a sending client MUST evaluate and observe. These properties can be informative or define what the sender of a message must follow in order for the delivery service to accept the message at all.

(***compatibility information:*** *it replaces the formerly defined mutableProfileExtension provided in the* `network.dm3.profile`)

Methode

```
// call to get a list of properties.
dm3_getDeliveryServiceProperties
```

Response

- **Message TTL:** Defines, how long the delivery service guarantees to cache a message. After the guaranteed time, the message MAY be removed, even if it was not picked up by the receiver. The minimum MUST be 30 days. If the value is not set, or set to 0 or null, messages will be cached unlimited (default).
- **Size Limit:** Each delivery service can define a maximum size of incoming messages. As the content of the message incl. attachments is encrypted for the receiver, the delivery service can't restrict attachments or other embedded data by its content. The only way is to restrict the total size of the message. A sender MUST check this property before sending the message, otherwise, the message may be rejected.

```
{
    // Number of days a delivery service must buffer a message
    // The message may be deleted if: incoming_timestamp_in_ms + days_to_ms(messageTTL)
↪< now_in_ms
    messageTTL: number;

    // The delivery service accepts only envelopes which fulfill the following␣
↪condition: sizeInBytes(envelope) <= sizeLimit
    sizeLimit: number;
}
```

In case of an error, an error object is returned as described in error codes.

2.3.3 Get the User's Profile Extension

Profile extensions are mutable properties of a receiver (identified by his/her ENS Name, . . .) that are not stored in the `network.dm3.profile` and can be changed without the need for a transaction. As these properties may vary between different delivery services, each delivery service to which the user is connected must provide this information.

Methode

```
// call to submit a message
dm3_getProfileExtension
```

Request

The request passes the **name** of the receiver.

```
// the name of the receiver
the_name
```

Response

The profileExtension contains configuration information of the receiver:

- **Encryption Scheme:** *(OPTIONAL)* The default encryption scheme is **x25519-chacha20-poly1305**. If another encryption scheme needs to be used (e.g., because this is needed for an ecosystem that is integrated into **dm3**), this can be requested. The default algorithm MUST be accepted, too. Otherwise, it might be impossible for a sender to deliver a message when it doesn't support the requested algorithm. This is a list of supported schemes, sorted by importance. All listed algorithms MUST be supported by the receiver. The sender is free to choose but should use receiver's preferences if supported.
- **Supported Message Types:** The receiver MUST provide a list of all **message types** that the client he/she uses is supporting (see message data structure). The message type **NEW** MUST be supported always and is set as default in case no information is delivered. The sender MUST NOT send unsupported messages, as the receiver will not accept those messages.

Other information (like spam protection settings) can be added to this struct (defined in protocol extensions).

```
{
  // Request of a specific encryption scheme.
  // (optional)
  encryptionScheme?: string[],
  // List of supported message types
  supportedMessageTypes: string[],
}
```

Example Profile Exception:

```
{
   "encryptionScheme": ["x25519-chacha20-poly1305"],
   "supportedMesssageTypes": ["NEW","EDIT", "READ_RECEIPT","RESEND_REQUEST"],
}
```

In case of an error, an error object is returned as described in error codes.

2.4 Appendix

2.4.1 Delivery Service

A Delivery Service is an RPC endpoint where a client can deliver its message (see *API*).

The delivery service

1. checks whether the message satisfies the recipient's spam policy (see ProfileExtension)
2. decrypts the envelope, adds a postmark, and stores the message encrypted for the recipient until the recipient picks it up.
3. if supported, notifies the receiver that there is a new message.

Delivery service nodes can be operated as a service or self-sovereign by the user. The protocol explicitly allows (see user profile) a user to point to multiple delivery services so that if one is not available, another can be used. However, delivery service nodes can also act as gateways to other protocols, services, or applications.

Gateway to other service

If a delivery service works as a gateway to another protocol or service, it must implement the *API* to receive dm3 messages. However, how it then processes the messages and delivers them to the ecosystem to which it is acting as a gateway is completely up to it and depends only on the service or protocol to which it is connecting.

A gateway can also provide multiple delivery service nodes as primary and fallback services.

2.4.2 Cross Chain Applications

Although the dm3 protocol is fundamentally designed to use ENS as a central registry, the protocol can also be implemented on other chains. Name services similar to ENS can be used as a local registry in the ecosystem of this chain. For the entire dm3 ecosystem, this local registry can be directly integrated via CCIP (Cross Chain Interoperability Protocol) (and vice versa) so that interoperability can be established. A complementary extension will further simplify the handling of names of such local registries by automatic mapping of top-level domains (see top-level alias at protocol extensions).

2.4.3 RPC Error Codes

Default error codes are specified in the JSON-RPC specification.

Error code	Error text	Description
-32600	Invalid Request	The JSON sent is not a valid Request object.
-32601	Method not found	The method does not exist / is not available.
-32602	Invalid params	Invalid method parameter(s).
-32603	Internal error	Internal JSON-RPC error.
-32700	Parse error	Invalid JSON was received by the server. An error occurred on the server while parsing the JSON text.

CHAPTER

THREE

MESSAGE STORAGE PROTOCOL (DM3MSP)

3.1 Storage

The dm3 protocol does not specify where the user has to store his messages. Depending on the use case and the requirements of the user, the messages can be stored in different places and ways. The user should always be in full control of his/her decision, where and how to manage conversation data.

The following approaches are possible, although not the only ones.

- **Local File Storage:** All conversations including attached media are stored in the local file system. All stored information is encrypted. While local storage is the most privacy preseving (conversation data is only stored an the user's device and in full control of the user), synchronization between devices is resistricted or not possible.
- **Web3 Decentralized Storage:** The conversations are organized and pinned in IPFS. The information is stored fully encrypted. Multiple devices of the user can access the decentrally managed data.
- **Cloud Storage:** The conversations are stored at a cloud service of the user (like Google drive). The information is stored fully encrypted. The user decides whether and/or which cloud service provider to use. The availability of the conversation data thus depends on the availability of the cloud service provider. Synchronization between different devices is easy as long as the cloud drive is accessable.
- **dm3 Service Storage:** A special variant of cloud storage is the data service of a dm3 node (delivery service). As an optional service, this can offer the storage of encrypted conversations. To access the conversation history, the client must be connected to this delivery service. Synchronization between multiple delivery service nodes is optional.

[IMAGE to visualize storage]

For performance reasons, a client can/should cache current conversations so that it does not have to fetch the data from a possibly remote storage each time it is accessed. If additional data is needed (e.g. earlier parts of the communication history), it can be retrieved sequentially.

3.2 Data Structure

The conversation data is stored as a data tree grouped by conversations in containers. The structure can be realized as a single file (e.g., export record), as a file store with multiple files (e.g., on local file system, a cloud drive, or in a decentralized storage), or as a database (e.g., as a service).

The data is structured in such a way that fast access with low overhead to specific data is possible, but retrieval of larger data packages is also efficient enough.

3.2.1 Architecture

The information containers are clustered in 3 types of data:

- **Root:** The root container contains a list of all conversation IDs. A conversation is specified as a collection of messages between 2 users. For each communication partner a conversation is stored. There is exactly one root container per user. The list of conversation IDs is encrypted (with the user's storage encryption key).
- **Conversations:** A conversation container contains a list of chunks, where the actual messages are stored. Each list entry, the chunk identifier, contains the chunk's id and the timestamp of the first message of the chunk. The list of chunks is encrypted (with the user's storage encryption key).
- **Chunks:** Each chunk container stores a list of messages. Those messages are sorted by timestamp. Each chunk can contain a different number of messages. The number is determined by the absolute size of the chunk. Chunks are encrypted (with the user's storage encryption key).

Root Container

The **key** of the `root` container is defined as SHA-256 hash of the signature of the user's ENS-name (represented by `$own_ens_name`).

```
key = sha256( signature( $own_ens_name ))
```

This might be the file name, database identifier, or the section name of a file.

The **conversations** list is the list of ENS-names of the conversation contacts which are the IDs of the conversations. These are ENS domains or ENS subdomains with **dm3** profile (see *registry*).

```
DEFINITION: Root Container

{
    conversations: string[]
}
```

This list is encrypted with the user's storage encryption key.

Example: (unencrypted)

```
{
   "conversations":
   [
      "friend1.ens",
      "0x123..abc.addr.dm3.eth"
   ]
}
```

Conversation Container

The **key** of a `conversations` container is defined as SHA-256 hash of the combination of the key of the root container `root.key` and the communication partner's ENS-name, which is listed in the `root` container's conversations list (represented by `root.conversations[$index]` at the `$index` of the list).

```
key = sha256( root.key + root.conversations[$index] )
```

For the same conversation partner, only one conversation can exist. A conversation must have at least one chunk with one message. Empty conversations are not stored.

A reference to a chunk is described by an identifier (incrementing, starting with 0) and the timestamp (unix time in milliseconds) of the first message in that chunk.

Note: *The timestamp can be used to find messages from a certain time period more easily without having to scan through all chunks.*

```
DEFINITION: Chunk Identifier

{
    // the chunk identifier, starting with 0
    id:   number,
    // timestamp of the first message in a chunk
    timestamp: number
}
```

The **chunks** list is the list of chunk identifiers describing the existig chunks. As chunks and messages are sorted in time, the last chunk in the list is the newest one and used to add a new message.

```
DEFINITION: Conversations Container

{
    chunks: ChunkIdentifier[]
}
```

This list is encrypted with the user's storage encryption key.

Example: (unencrypted)

```
{
   "chunks":
   [
      { "id": 0, "timestamp": 16759549450000 },
      { "id": 1, "timestamp": 16762446650000 }
   ]
}
```

Chunk Container

The **key** of a chunks container is defined as SHA-256 hash of the combination of the key of the conversations container $conversation.key and the stringified chunk identifier, which is listed in the conversation's container's chunks list (represented by $conversation as the current conversations container and the chunk identifier $conversation.chunks[$index] at the $index of the list).

```
key = sha256( $conversation.name + stringify( $conversation.chunks[$index] ))
```

The **messages** list contains the envelopes (see Encryption Envelope Structure) including the message. However, the fields encrypted during transmission are decrypted. The entire chunks container is subsequently encrypted with the storage encryption key.

```
DEFINITION: Chunk Container

{
    messages: Envelope[]
}
```

The size of the chunks is defined by the maximum size. The maximum size is defined by the client.

Recomentation:

- minimum: 500kB,
- maximum: 10MB.

A chunk must contain at least one message. Empty chunks are not created.

Note: *The envelope with the message contains encrypted content during transmission (end-to-end encryption). The decrypted information is used for storage so that the sender's public key is not required again to decrypt the data for future use.*

Note: *The maximum size of a chunk must be bigger or equal to the maximum size of a message defined by the delivery service (see maxium message size).*

The messages list is encrypted with the user's storage encryption key.

Example: (unencrypted)

```
{
   "messages":
   [
      { "message": {...}, "metadata": {...}, ...},
      { "message": {...}, "metadata": {...}, ...}
   ]
}
```

CHAPTER

FOUR

REFERENCES

[NIR1] Nir, Y.; Langler, A.: **ChaCha20 and Poly1305 for IETF Protocols.** Internet Research Task Force (IRTF), Request for Comments: 7539, ISSN: 20070-1721, https://datatracker.ietf.org/doc/html/rfc7539.

[RPC1] JSON-RPC Working Group: JSON-RPC 2.0 Specification, Origin Date: 2010/03/26, updated 2013/01/04, https://www.jsonrpc.org/specification.

CHAPTER FIVE

INDICES AND TABLES

- genindex
- modindex
- search