ChatGPT

EMSG Protocol Specification (v0.2)

1. Overview

EMSG is a decentralized, federated messaging protocol designed for secure, interoperable chat across domains. Its purpose is to enable real-time communication without central servers, giving users control over their data and identities. EMSG's goals include **secure messaging**, **global federation**, **extensible metadata**, and **compatibility**. Each user and domain runs its own server software, and servers interoperate via standard protocols. In EMSG, users are identified by unique handles (`user#domain.com`) and communicate over encrypted channels. The protocol uses DNS for discovery and JSON for message formatting, similar to XMPP or ActivityPub: for example, XMPP identifies users as `username@domain` in a decentralized network [1]. EMSG builds on these ideas to define addressing, discovery, transport, and message structures for a modern federated chat system.

2. Addressing Format

Users are addressed as `user#domain.com`. The **domain** part designates the home server (authority) for that user. This ensures global uniqueness: each domain manages its own namespace and guarantees that `user#domain.com` identifies exactly one account. This approach parallels email or XMPP addressing [1]. The `#` separator is chosen to distinguish EMSG IDs from email addresses. Each username must be unique on its domain, and the combination of username and domain forms the globally unique identifier. For example, if Alice has an account on `example.com`, her EMSG address might be `alice#example.com`. Domains can also support lookup via webfinger or `.well-known` (out of scope here), but fundamentally each user ID contains the domain that can be looked up via DNS.

3. DNS Discovery

EMSG uses DNS records to locate the messaging server for a domain. Each domain should publish an **EMX** record (an "EMail-eXchange"–like record) analogous to an MX record for email, pointing to the host running the EMSG service. In addition, DNS SRV records are used for flexible service discovery, much like XMPP does for chat servers [2]. For example, a domain `example.com` might have:

- An A/AAAA record: `example.com → IP of the EMSG server.`
- An SRV record: `_emsg-client._tcp.example.com. IN SRV 10 5 5888 emsg1.example.com.`
- An SRV record: `_emsg-server._tcp.example.com. IN SRV 10 5 5888 emsg1.example.com.`

These SRV records tell clients (port 5888 by convention) and other servers how to connect. Like XMPP's use of `_xmpp-client` and `_xmpp-server` SRV records, EMSG uses its own service names to redirect connections [2]. The SRV priority/weight fields allow multiple servers for load balancing or failover. If no SRV is found, clients may fall back to the A record. Proper DNSTTL values should be set to balance caching and update agility. DNSSEC/DANE may be used to secure these records against tampering.

4. Transport Protocol

EMSG defines **ESMP** (EMSG Session Messaging Protocol) running over TCP. The standard port is **5888/TCP** by default. All client-to-server and server-to-server traffic is carried as line-delimited JSON messages over a persistent TCP connection (optionally protected by TLS). Each JSON message is framed (e.g. newline-terminated) to ensure clear boundaries. Messages include a JSON object with fields such as `type`, `sender`, `payload`, etc. The use of JSON provides a lightweight, text-based data format [3]. For example, a chat message might be:

```
{ "type": "chat", "sender": "alice#example.com", "to": "bob#example.org",
  "timestamp": 1610000000000, "content": "Hello!" }
```

Before exchanging chat messages, the client and server perform a handshake (e.g. exchanging a `hello` message including protocol version and capabilities). Servers may require TLS for encryption and may present certificates for domain authentication. Message payloads and any attachments are encoded within JSON or as separate binary protocols (outside the scope here). ESMP is intentionally simple: each message is a JSON object with well-known fields, ensuring interoperability.

5. Message Types

EMSG supports several message types for user communication and system events. Each message JSON has a `type` field indicating its category. The two main types are:

- **Chat**: User-to-user chat messages, either one-to-one or group chat. A chat message carries a text or media payload. Fields include `content`, `timestamp`, `sender`, `recipient` or `group_id`, etc. Example:

  ```
  { "type": "chat", "sender": "alice#ex.com", "recipient": "bob#org.com",
    "timestamp": 1610000100000, "body": "Hi Bob!" }
  ```

- **System**: Protocol/system messages for events and actions. System messages notify clients of changes in group membership, administrative actions, or profile updates. Subtypes of `system` include:

- `group_created`: A new group was created (includes `group_id` and initial `group_name`).
- `joined`: A user joined a group.
- `left`: A user left a group voluntarily.
- `removed`: A user was removed (kicked) from a group.
- `admin_assigned`: A user was granted admin privileges in the group.
- `admin_revoked`: A user's admin role was revoked.
- `profile_updated`: A user updated their profile (e.g. name or avatar changed).
- `group_renamed`: The group's name was changed.
- `dp_updated`: The group's display picture (DP) was updated.
- `description_updated`: The group's text description was changed.

These system events typically include the relevant user or group IDs and timestamp. For example, a group creation event might look like:

```
{ "type": "system", "event": "group_created", "group_id": "grp123",
  "group_name": "Project Team", "creator": "alice#ex.com", "timestamp":
1610000200000 }
```

The specific names and payload fields follow a consistent JSON schema. Many existing messaging frameworks use similar event notifications – for instance, ZEGOCLOUD's IM generates tip messages for group events such as `GROUP_CREATED`, `GROUP_JOINED`, `GROUP_LEFT`, etc. [4]. Likewise, group profile changes (name/avatar) are signaled separately [5]. EMSG unifies these into explicit system event types so that clients can display or log them.

6. Group Messaging

Groups in EMSG are identified by a persistent `group_id` (a unique identifier, often a UUID or opaque string). This serves as the thread identifier for the group chat. Once created, `group_id` remains constant so that chat history and membership can be tracked. Group metadata includes:
- **group_name**: A human-readable title for the group.
- **description**: An optional text description or purpose of the group.
- **display_picture**: A URL or image reference for the group's icon.

For example, a group metadata JSON might be:

```
{ "group_id": "grp123", "group_name": "Project Team",
  "description": "Chat for project members",
  "display_picture": "https://ex.com/img/teams/proj.png" }
```

When a group is created, this metadata is set (and can later be updated via system messages like `group_renamed`, `description_updated`, or `dp_updated`). Group metadata should be stored by the server and optionally synchronized to members. This is analogous to room names and avatars in XMPP MUC or metadata in ActivityPub collections. The group ID functions like Matrix's `room_id` – a persistent identifier for the conversation thread [6].

7. User Profile

Each user has a profile containing personal fields. EMSG profiles may include:
- **first_name**, **middle_name**, **last_name**: Components of the user's real name. (E.g. first and last names, with middle name optional.) This maps to schema.org's `givenName`, `additionalName`, `familyName` fields [7] [8].
- **display_picture**: A URL to the user's avatar or photo. This corresponds to schema.org's `image` property [9].
- **address**: The user's physical address or location. (Text or structured address.) Schema.org defines `address` for postal addresses [10].

Each profile field may have visibility controls (public, friends-only, private). For example, a user can choose to share only first and last name publicly, but hide address. The server respects these settings when responding to profile queries. Profile updates trigger `profile_updated` system messages so contacts can refresh cached info. In summary, the profile schema closely follows standard vocabularies

for person data [7] [9], allowing fields to be optional or required as needed (e.g. first and last name required, middle name optional).

8. JSON Schema Examples for Messages and Metadata

EMSG uses JSON Schema to define the structure of messages, profiles, and group data. JSON Schema ensures data consistency and validity [11]. For example:

- **Chat Message Schema (example)**:

```
{
  "type": "object",
  "properties": {
    "type": { "const": "chat" },
    "sender": { "type": "string", "format": "hostname" },
    "recipient": { "type": "string", "format": "hostname" },
    "group_id": { "type": "string" },
    "timestamp": { "type": "integer" },
    "body": { "type": "string" }
  },
  "required": ["type", "sender", "timestamp", "body"]
}
```

This schema enforces that every chat message has a `type` of `"chat"`, a `sender` ID, a `timestamp`, and a text `body`. Optional fields like `recipient` or `group_id` indicate one-to-one versus group chat.

- **User Profile Schema (example)**:

```
{
  "type": "object",
  "properties": {
    "first_name": { "type": "string" },
    "middle_name": { "type": "string" },
    "last_name": { "type": "string" },
    "display_picture": { "type": "string", "format": "uri" },
    "address": { "type": "string" }
  },
  "required": ["first_name", "last_name"]
}
```

This schema defines the fields a user profile can contain. `middle_name`, `display_picture`, and `address` are optional. A profile update message might include this profile object.

- **Group Metadata Schema (example)**:

```
{
  "type": "object",
  "properties": {
    "group_id": { "type": "string" },
    "group_name": { "type": "string" },
    "description": { "type": "string" },
    "display_picture": { "type": "string", "format": "uri" }
  },
  "required": ["group_id", "group_name"]
}
```

This schema covers the group properties. EMSG implementations should validate outgoing and incoming JSON against their schemas to avoid malformed data. Using JSON Schema provides a clear contract for developers and enables shared understanding across servers [11].

9. Security

Security in EMSG is layered:

- **Message Signing and Verification**: Every message (chat or system) is digitally signed by the sender's private key. The signature (e.g. Ed25519 or RSA) is included in the message envelope. Recipients verify signatures against the sender's public key. Ed25519 is recommended for compact, high-speed signing [12]. RSA (with PKCS#1 padding) is also supported for interoperability [13]. For example, a message object may include a `"signatures"` field with the algorithm (e.g. `ed25519`) and signature value. This ensures authenticity and integrity: servers reject messages whose signature does not match the claimed sender.

- **End-to-End Encryption (Optional)**: Message content can be encrypted end-to-end on the client side. EMSG supports integration with double-ratchet protocols (like OMEMO) or other E2EE schemes. For instance, an encrypted chat payload might use the Double Ratchet Algorithm to achieve forward secrecy and offline message delivery [14]. Encryption is optional and negotiated per conversation. When enabled, only recipients with the proper keys can decrypt the `body` of a chat message; metadata (sender, group, timestamp) remains visible to route the message.

- **Field-Level Visibility and Access Control**: Sensitive profile or group fields can be hidden or encrypted. For example, a user's address might only be shared with contacts, or a group's description visible to members only. The protocol allows implementing "read ACLs" per field. In practice, the server may only include certain fields in profile query responses based on the requestor's permissions. More advanced implementations could encrypt individual JSON fields such that only certain clients (e.g. group admins) can decrypt them.

- **Authorization**: Certain actions (e.g. kicking a user, renaming a group) are restricted to authorized roles. EMSG defines that only group admins (or owners) may perform administrative changes. Servers must enforce these rules: a `admin_assigned` event can only be sent by a group owner, for example. Field updates to user profiles can be done only by the user themselves (or delegated authorities). All clients should verify that incoming system messages are from a trusted server or user to prevent spoofing.

By combining signatures, optional encryption, and robust ACLs, EMSG provides a secure messaging framework. Standard cryptographic practices (keys, nonces, certificates) should be used as described in the relevant RFCs.

10. Error Codes

EMSG uses standard numeric error codes to indicate success or failure of requests, similar to HTTP status codes [15]. Each response message may include a `code` and `message`. Typical code ranges:
- **1XX (Informational)**: e.g. `100 Continue` (request received, proceed).
- **2XX (Success)**: e.g. `200 OK` (request succeeded), `201 Created` (resource created).
- **4XX (Client Error)**: e.g. `400 Bad Request` (malformed JSON or missing field), `401 Unauthorized` (bad credentials), `403 Forbidden` (action not allowed), `404 Not Found` (unknown user/group).
- **5XX (Server Error)**: e.g. `500 Internal Error` (unexpected failure), `501 Not Implemented` (feature not supported).

For instance, if a client tries to send a message to a non-existent user, the server might reply with `{ "code": 404, "message": "User not found" }`. Using established semantics (e.g. `200`, `400`, `403`) helps clients handle errors consistently. Implementations should document any additional EMSG-specific error codes.

11. Versioning

EMSG is versioned (here v0.2). Each message exchange can include a `version` field (e.g. in the handshake or message header) to negotiate compatibility. We follow **Semantic Versioning (SemVer)** for protocol versions [16]. Version numbers take the form `MAJOR.MINOR.PATCH`: - **MAJOR** is incremented for incompatible changes. A client/server must reject communication if its major version does not match the peer's.
- **MINOR** is incremented for backward-compatible feature additions. Clients should handle unknown new fields gracefully (ignoring unsupported ones).
- **PATCH** is for backward-compatible bug fixes.

For example, in the initial handshake, the client might send `"version": "0.2.0"`. If a server speaks v0.1, it may still interoperate if only minor/patch differences exist, but if the client were v1.0, the server would refuse (`411 Unsupported Version`). Proper version negotiation ensures that new features can be rolled out without breaking older clients [16].

12. Federation and Decentralization

EMSG is designed for a **federated architecture**. Each domain's EMSG server communicates with other domains' servers to deliver messages. Domains interoperate by addressing users cross-domain (`user#domain.com`) and by following the common ESMP protocol. This is similar to XMPP's federated servers or the decentralized social web (ActivityPub) model. For instance, XMPP is described as a "decentralized network" where each user is identified by `username@domain` and domains route messages to each other [1]. Likewise, ActivityPub's server-to-server federation allows separate websites to share posts in a common network [17]. In EMSG, when a user on `a.com` sends a message to `bob#b.com`, `a.com`'s server does a DNS lookup (SRV/EMX) for `b.com`, establishes a TCP connection on port 5888, and exchanges messages in ESMP format. Each server only vouches for its own users; for other domains it verifies signatures on incoming messages to trust their authenticity. Because domains independently operate, there is no single point of failure. New domains can join by implementing the

protocol and publishing DNS records. Federation also means metadata (like user public keys, profile, etc.) may be propagated in a distributed manner (e.g. a server can query `bob#b.com` for Bob's public key via a special `GET_PROFILE` message). Overall, EMSG's federation model follows proven patterns of decentralized messaging [1] [17].

13. Implementation Prompts for Agentic IDEs

To assist developers and intelligent IDE tools, the following implementation prompts are suggested:
- **Schema Evolution**: *Add/Remove Fields* – When adding new message or profile fields, update the JSON Schema accordingly. Remove deprecated fields carefully, ensuring compatibility rules (versioning).
- **Schema Update Handling**: *Update Schema* – Provide clear migration paths. For example, version v0.3 adds a new `nickname` field to profiles. The IDE could suggest default values for older records or show diffs between schema versions.
- **Profile Change Propagation**: *Handle Profile Changes* – When a user updates their profile, the server should generate a `profile_updated` event. Implement logic to push or notify contacts of this change.
- **Group Change Management**: *Handle Group Metadata Changes* – Changes to group (name, description, DP) should emit corresponding system messages (`group_renamed`, `description_updated`, `dp_updated`). Clients should update UI accordingly.
- **Permission Checks**: *Enforce Access Control* – On events like admin assignment or user removal, ensure only authorized users (existing admins) can trigger them. The IDE can flag missing permission checks.
- **Error Handling**: *Standardize Errors* – Maintain a list of error codes. If a new validation rule is added, define appropriate error codes/messages.
- **Testing Interoperability**: For each feature added (e.g. `dp_updated`), create tests across different implementations to confirm consistent handling.

These prompts can guide automated tools to update the protocol library, validation schemas, and client logic when the EMSG specification evolves.

14. Roadmap

EMSG v0.2 will be rolled out in phases:
1. **Phase 1 – Internal Testing**: Implement basic server and client with core features (addressing, simple chat, DNS discovery, signing). Test within a controlled network.
2. **Phase 2 – Alpha (Limited Federation)**: Deploy on a few domains. Enable group chat and system events. Collect feedback and fix bugs.
3. **Phase 3 – Beta (Wider Release)**: Open to public testers. Add optional features like E2EE. Monitor scaling and federation issues.
4. **Phase 4 – Production v1.0**: Declare the protocol stable (v1.0.0) after ironing out issues. Provide libraries and documentation for developers.

A *phased rollout* mitigates risk by incrementally implementing the system [18]. Early users test functionality and provide feedback, helping refine later phases. At each phase, we update documentation and schema. The version number is bumped as per SemVer (e.g. moving from 0.2 to 1.0 when reaching stability).

All sections above are normative parts of the technical specification and must be adhered to by compliant EMSG implementations.

Sources: Design ideas and practices are adapted from established federated messaging standards and specifications [1] [3] [4] [7] [12] [15] [16] [17] [18].

---

[1] [2] DNS configuration in Jabber/XMPP – Prosody IM
https://prosody.im/doc/dns

[3] RFC 8259 - The JavaScript Object Notation (JSON) Data Interchange Format
https://datatracker.ietf.org/doc/html/rfc8259

[4] [5] In-app Chat(Web) Receive tip messages
https://www.zegocloud.com/docs/zim-web/guides/messaging/receive-tip-messages

[6] Room Version 4 | Matrix Specification
https://spec.matrix.org/v1.10/rooms/v4/

[7] [8] [9] [10] Person - Schema.org Type
https://schema.org/Person

[11] JSON Schema
https://json-schema.org/

[12] RFC 8032 - Edwards-Curve Digital Signature Algorithm (EdDSA)
https://datatracker.ietf.org/doc/html/rfc8032

[13] RFC 8017 - PKCS #1: RSA Cryptography Specifications Version 2.2
https://datatracker.ietf.org/doc/html/rfc8017

[14] OMEMO - Wikipedia
https://en.wikipedia.org/wiki/OMEMO

[15] RFC 7231 - Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content
https://datatracker.ietf.org/doc/html/rfc7231

[16] Semantic Versioning 2.0.0 | Semantic Versioning
https://semver.org/

[17] ActivityPub
https://www.w3.org/TR/activitypub/

[18] What is phased rollout? | Definition from TechTarget
https://www.techtarget.com/searchitoperations/definition/phased-rollout