# *fmsg*

## A message exchange protocol

Mark Mennell
@markmnl@fmsg.org
2026

**Abstract.** A message definition and protocol for sending and receiving messages. Compared to email's SMTP and IM applications, fmsg has many overlapping features; like email, fmsg is an open protocol anyone can host, unlike email but like IM applications fmsg is efficient at conversational message exchanges. Included in the fmsg design are security features built-in giving sender verification and message integrity. Sent via a fmsg host to one or more recipients, each message in a thread is linked to the previous using a cryptographic hash. Two properties arise from the design: proof of prior participation and eventual verifiability. Real world tests comparing fmsg to email are presented  Overall the case is made that fmsg provides an efficient and secure messaging system with ownership and control at the domain level with less operational complexity because of the guarantees built-in.

# Table of Contents

# Introduction

Electronic mail (email) has co-existed with The Internet since the Internet's early development and has become ubiquitous for messaging, from personal to corporate and even official communications. Meanwhile, Instant Messaging (IM) applications have proliferated with businesses building products with ever-increasing features. Many such applications have become part of everyday life. Despite the advancement of IM applications, email still maintains a stronghold for certain types of communication.

This paper explores successes and issues with the everyday messaging applications we use – email and IM applications; fmsg (thought of as “f-message”, always stylised lower-case) is then introduced – a binary protocol intended as an alternative to both. A feature-wise comparison is made between email, IM applications and fmsg. A real-world comparison is made by sending and receiving various messages using fmsg and email.

Lastly, an epilogue is included discussing the use case of fmsg for AI agents – specifically, computer programs utilising large language models (LLMs) as reasoning steps through messages sent to and received to and from a LLM. Appended are links to relevant code and document repositories, lists of IM applications and a snapshot of the full fmsg specification (so this paper can be read standalone).

# Stronghold of Email

Email remains widely used today because it offers ownership and control at the domain level: messages are routed using Domain Name System (DNS) defined mail exchangers, enabling server-to-server communication and allowing domain owners – especially corporations – to manage their own mail infrastructure and meet requirements around privacy, data sovereignty, cost, and legal obligations. Since email is an open and decentralised standard, anyone can run their own email service, while personal users overwhelmingly rely on hosted providers to avoid the complexity of deploying and operating an email service.

Email also carries a sense of formality due to its richer formatting options and its asynchronous, non-conversational nature. Another reason for email’s persistence is that an email address often forms the basis of an online identity – online services use email for verification and communication with a user creating an account. Consequently, many online services require or commonly request an email address when creating an account.

# Pain of Email

The success and massive adoption of email over decades has inevitably led to issues, which fmsg aims to solve, namely: **spam**, **efficiency**, operating **complexity** and **user experience** – particularly with back-and-forth conversation. Spam, or “junk” email, accounted for 45.6% of global email

traffic in 2023[1]. Apart from the annoyance, spam consumes computing resources, especially in filtering out junk. Email uses the Simple Mail Transfer Protocol (SMTP) for server-to-server transmission, with clients also using SMTP to send messages to their server, then protocols like Post Office Protocol (POP) and Internet Message Access Protocol (IMAP) to retrieve them. Email clients typically include the entire original content, causing long email "chains," or "threads," to grow quickly in size and making email unsuitable for conversational style communication. User experience suffers because the formatting of included original content lacks a consistently implemented standard, meaning different email clients use varying formatting (e.g., prefixing lines with ">" or "|" and placing original content above or below new content), sometimes failing to correctly understand other clients' formatting when attempting to display threads in their natural hierarchy. Because traditional email represents binary attachments using MIME content-transfer encodings such as Base64, this increases their size by about one third. These efficiency and user experience issues become more and more egregious as replies build up, especially when some replies branch off the main thread.

# Instant Messaging Applications

Along with the rise in online connected devices, IM apps have proliferated as a way to electronically message someone else. IM apps tend to send their messages immediately to central server(s) owned and operated by the companies that made them. The message is then sent on to the recipient, who, if online, will usually receive it almost immediately, hence "instant" in the name. IM apps evolve at the pace of their creators and contain numerous differentiating features compared to other IM apps. However, they largely follow this client-server architecture whereby the owner handles the messages and their data.

The top ten personal IM apps at the time of writing, based on popularity, are listed in Appendix II – IM Applications. WhatsApp has the highest number of monthly active users, with over three billion, while Weixin and WeChat combined exceed 1.4 billion. Privacy-focused IM apps like Telegram and Signal have also achieved substantial adoption. Some IM applications are popular only in particular regions of the world, such as KakaoTalk in South Korea and LINE in Thailand, Taiwan and Japan.

## Group Messaging Platforms

Missing from the list of popular IM apps is a class of messaging applications that have become increasingly prevalent in workplaces and online communities, such as Slack, Microsoft Teams and Discord. These instant messaging platforms centre around groups of users (in addition to one-on-one messaging, often called "direct message" or just "DM"). We refer to these here as "Group Messaging Platforms". Technically, these messaging platforms also follow a client-server architecture. For the sake of completeness, we list these applications too in Appendix II – IM Applications. For the purposes and intent of this paper we consider these platforms included in "IM

1 DeBounce, "Email Spam Statistics 2026," debounce.com, October 2025. https://debounce.com/blog/email-spam-statistics/ Accessed 25 Mar 2026

apps", with the exception of self-hosted platforms that are capable of being hosted on one's own infrastructure, such as Zulip, Rocket.Chat and Mattermost. This is an important distinction because when self-hosted, owners retain control over their infrastructure and data.

One notable exception among the messaging systems discussed so far is Matrix, described as "a set of open APIs for decentralised communication"[2]. Like these other Group Messaging Platforms, Matrix is concerned with instant messaging between groups of users. Unlike those platforms, Matrix is decentralised in that self-hosted servers – each defining their own users – can join (federate) and communicate with users from other Matrix servers. Matrix is more of an ecosystem than an application, so is quite dissimilar to the "IM apps" we refer to generally throughout this paper.

The fundamental difference between these Group Messaging Platforms, including self-hosted platforms like Matrix, and fmsg is that *fmsg is just messages* – there are no forums, rooms or channels, i.e. groups of users. In fmsg, to receive a message someone has to send it to you. As such, fmsg has no management around group membership or synchronisation of messages in those groups. In this regard fmsg is more similar to email. However, unlike email, in which messages are standalone, threads in fmsg messages can organically evolve into group-like chats.

## Features of fmsg

Perhaps the best way to introduce fmsg is a feature-level comparison to email and IM apps. The following features are plotted in the Venn diagram below. Note that some features deliberately bleed across boundaries, indicating their partial inclusion – for instance, "prone to spam" is mostly in email and to a lesser extent in fmsg, while "rich text formatting" is wholly in email and fmsg and only partially in IM apps.

*Figure 1: Feature comparison*

**Attached files** – Arbitrary files can be included with messages. Email and fmsg include arbitrary file attachments; some IM apps do too, though often with limitations on the types or sizes of files.

**Audio and video calls** – Live audio and video transmission, typically supported by IM applications and generally implemented using protocols separate from the messaging protocol, though initiated and bundled within the application. Included here more to highlight that live streaming is not a use case of fmsg.

**Conversations friendly** – Refers to the back-and-forth message replies between two or more participants. Email is poorly suited to conversational-style communication because of message growth (email clients commonly include the full quoted message history in subsequent replies) and latency introduced at several stages in email's architecture, such as queuing, relaying and spam filtering. Email can still arrive quickly, but its store-and-forward design does not provide the same immediacy as IM applications, where participants can exchange messages in near real time. fmsg is

2 Matrix.org Foundation, "Matrix specification v1.17", 2026, https://spec.matrix.org/v1.17/, accessed Mar 25, 2026

also store-and-forward, like email, but messages are sent directly between hosts and use compact binary encoding. Messages reference previous messages rather than re-transmitting their content, so where a referenced message has already been accepted by the recipient, its content need not be sent again. This makes fmsg suitable for near-real-time conversations between participants as well.

**Message integrity** – Whether messages can be verified as unaltered in transit and at rest. Email itself does not provide a universal message-integrity guarantee. Various additional mechanisms can provide particular protections: Transport Layer Security (TLS) protects communications in transit between parties, DomainKeys Identified Mail (DKIM) cryptographically signs selected message content and headers, and Domain-based Message Authentication, Reporting & Conformance (DMARC) provides domain-level authentication and policy. These mechanisms contribute to the security of email but do not provide a single, inherent message-integrity property.

fmsg has message integrity built-in, allowing a delivered message to be independently verified as unaltered even after delivery. Proprietary IM applications presumably implement message-integrity mechanisms, but these cannot be independently verified by users or third-party implementations.

**Multimedia messages** – Pre-recorded audio and video files – a subset of arbitrary file attachments – are generally supported by email, fmsg and IM apps.

**Ownership and control** – Who has authority over the infrastructure, user accounts and message handling – where and how is the data stored? Both email and fmsg can provide ownership and control at the domain level, whereas IM applications typically handle users data for them.

None of these systems are inherently peer-to-peer between individual users. Email and fmsg can be operated without a third-party by running one's own infrastructure. Operating one's own email server has become increasingly impractical due to the complexity of operating a reliable service and the extent of anti-spam measures that email servers must enforce, driving widespread adoption of third-party providers such as Gmail, Outlook and Yahoo – centralising email service providers. One of the motivations of fmsg is to enable organisations, and even individuals, to own and operate a messaging service without relying on third parties to handle their messages[3].

**Prone to spam** – Whether an implementation is exposed to undesired messages typically sent in bulk for commercial, fraudulent or malicious purposes. Any service that permits unsolicited messages is exposed to this problem. Email is particularly exposed because of its wide adoption – a victim of its own success – together with the relatively low cost of sending email and the lack of anti-spam measures in its initial design. IM applications have more control such as who can initiate contact.

3 Personally I have self-hosted both email and fmsg, I have found that both require registering a domain, configuring DNS records and having a stable, reachable, public IP address. Where they differ, as a residential customer, my ISP would not allow me to configure the reverse DNS (PTR) record for my public IP address – used by email for anti-spam controls – so had to use an email relay for reliable delivery – which concerned me because email relays can read the content of one's email! fmsg's direct host-to-host requirement prevents relaying and has no requirements outside my control, like the reverse DNS record, making self-hosting fmsg more viable and secure for me than email.

fmsg does allow unsolicited messages and is therefore exposed to spam. However, several properties of fmsg provide useful signals for mitigating spam. Built-in sender verification makes the sender proven, combined with the senders' history and proven participation in threads can be used in filtering decisions. Also, the challenge-response mechanism discussed later requires the sending host to be an active participant during message exchange – to respond to challenges and compute message digests.

**Rich text formatting** – Formatted text messages, e.g. fonts, font size, emboldening, tables, lists etc., to provide control over presentation. Practically, email supports a subset of HyperText Markup Language (HTML) for rich text formatting. Both fmsg and email allow arbitrary media types, but clients determine what to render inline. IM applications vary considerably in their support for formatting, from plain text to richer formatting and application-specific presentation features.

**Sender Verification** – Verification that messages are from who they claim to be from. SMTP itself does not verify the sender is genuinely the originator, so the sender can be spoofed. Additional mechanisms are therefore needed, such as Sender Policy Framework (SPF), which checks whether a sending host is authorised for a domain. DKIM provides cryptographic authentication for signed message content and headers, while DMARC provides domain-level authentication and alignment policy.

fmsg includes domain-level sender verification built in – the sending host's IP address must be authorised by the domain, and the senders host can be challenged to prove it indeed has the message being transmitted.

**Text messages** – Plain text messages are supported by all.

**Unsolicited messages** – Anyone can send you a message given your address, supported by email using an email address and fmsg using an fmsg address. Some IM applications also support unsolicited messages, often using a verified phone number or another user identifier. However, mainstream IM applications are generally closed systems in that they only support messaging between users of the same platform – you cannot send a message between WhatsApp and LINE like you can send email between Gmail and Yahoo.

# Tour of fmsg

The full fmsg specification is appended consisting of the message definition – defining messages data structure, and the protocol – describing the sequence of steps and rules to follow when transmitting a message. A message consists of header fields (size, topic, type etc.) including a cryptographic digest (SHA-256) of a parent message, if any. Messages are sent via a sender's fmsg host to one or more recipients by sending the message to each distinct domain from the recipients list. As message threads, or "chains", are created and branch over time, a hierarchical structure builds up with each reply – a rooted tree (a type of directed acyclic graph, or, DAG). During message exchange the receiving host can challenge the sending host for detail of the message being sent on a new connection.

## Addresses

An fmsg address looks like an email address except has a leading “@” character.

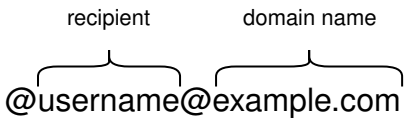


*Figure 1: An fmsg address*

Domain part is the domain name owning the address. Recipient part identifies the recipient (case insensitive) known to hosts for the domain. A leading “@” character is prepended to distinguish from email addresses. The secondary “@” separates recipient and domain name as per norm. Recipient part is a string of characters which can be any letter in any language, any number, the hyphen "-" or underscore "_" characters non-consecutively and not at beginning or end. Refer to the specification in Appendix III for the full description.

## The DAG


*Figure 2: Message Thread Example*

Having reference to the previous message with each reply allows deterministic construction of a message thread's DAG by all participants. Previous messages in a thread need not be included each subsequent reply. Knowledge of prior messages in a chain enforces senders and recipients have the previous message (or at least the hash) because when computing a new message's hash you need the previous message's hash (the pid field). Without *a* message in a thread there is no way to join the thread because you need the previous message. Also, during message-exchange, the protocol enforces the sender is a participant of the parent message being replied to.

Existing participants who may have lost their messages, or new participants joining a thread, can be added by someone, using the add-to functionality on any message, enabling replies to that message of the previous participants plus the new additions.

*To respond in a thread and append to the graph someone has to be participant of the message they are responding to.* Prior participation is therefore required except on the first message starting a thread. Therefore to have replied we can, we call this: proof-of-prior-participation.

## The Challenge

A defining feature of the fmsg protocol, that builds on and improves sender verification and message integrity, is the automatic challenge. The challenge is optional at the receiver's discretion, so the sender always needs to be ready to respond to ensure successful message delivery. Reasons a receiving host may choose not to challenge is to trade efficiency for protocol guarantees such as when the sending host is on the same domain, or, the sender was a prior participant in the message thread of the message being sent (i.e. proof of prior participation).

The challenge is a separate connection opened by the receiving host to the sending host during receipt of a message before the incoming message content will be downloaded and accepted. The challenge message is sent to the remote fmsg host which in turn replies with SHA-256 hash of the entire message being sent, later used to verify the entire message once downloaded.

A brute force style attack where many messages to many hosts could trigger challenges to an unsuspecting host (if the messages spoof an address belonging to the unsuspecting host), is prevented by the protocol requirement to verify the origin IP address is authorised before issuing a challenge. The receiving host performs a lookup on the subdomain "fmsg" of sending fmsg address, e.g. fmsg.example.com if the sender was @alice@example.com, to ensure the IP address of the sender's address is known to sender's domain. If the IP address is not authorised the message exchange is terminated.

This separate challenge request provides some mitigation to "spamming" – because not only is the sender verified (at the domain level) – which helps source domain reputation tracking – but also requires:

1. listening for an incoming connection,
2. computing, retaining and lookup of outgoing messages digests; and,
3. responding to the challenge.

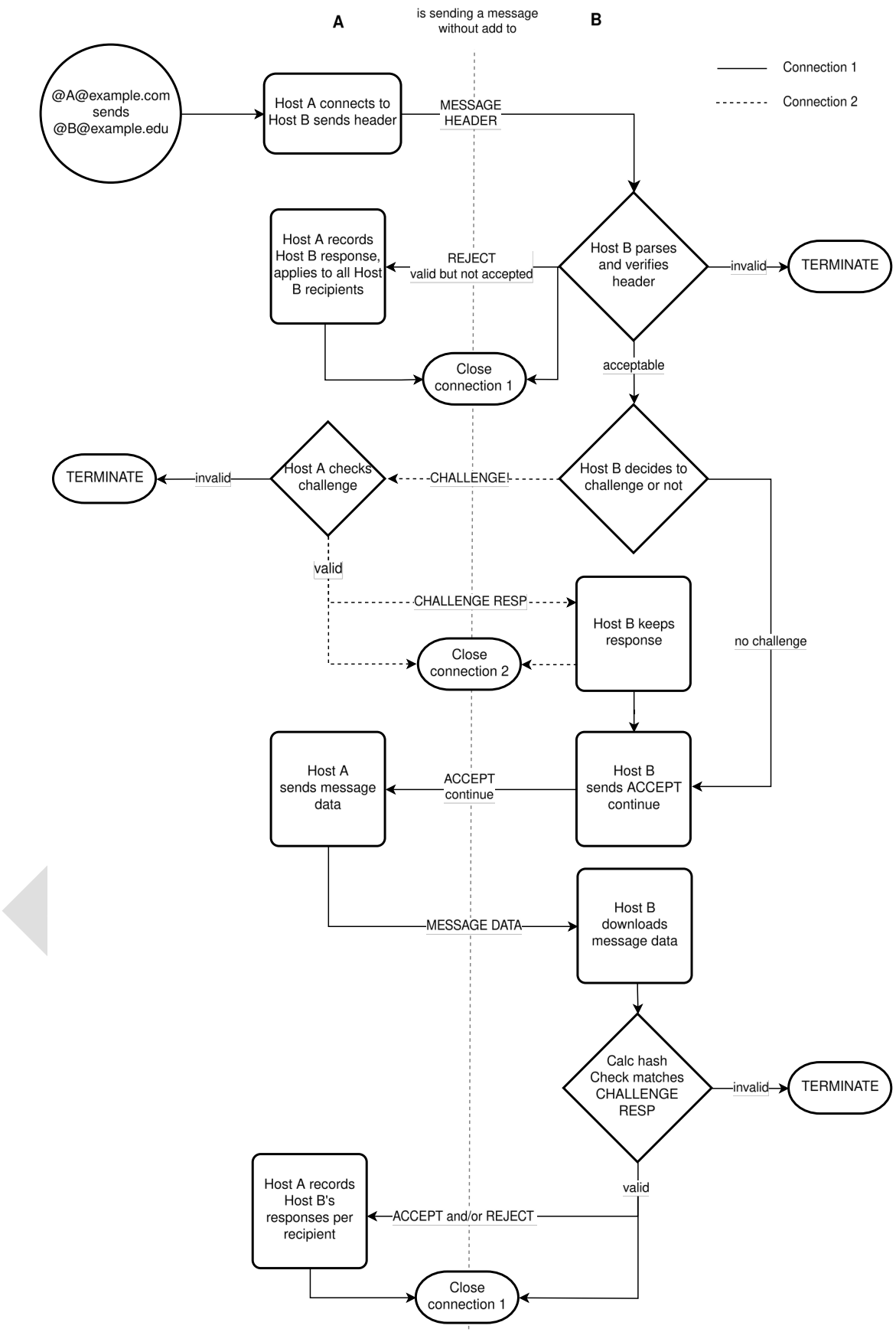


*Figure 3: Protocol flow diagram*

# Practical Considerations

fmsg itself only describes host-to-host communication – leaving other functionality such as retrieval of messages, user identity and access management, undefined. So a full fmsg setup includes other components which would vary between deployments and use-case. For instance an enterprise identity management system could be integrated on one host, and a custom user management system on another host.

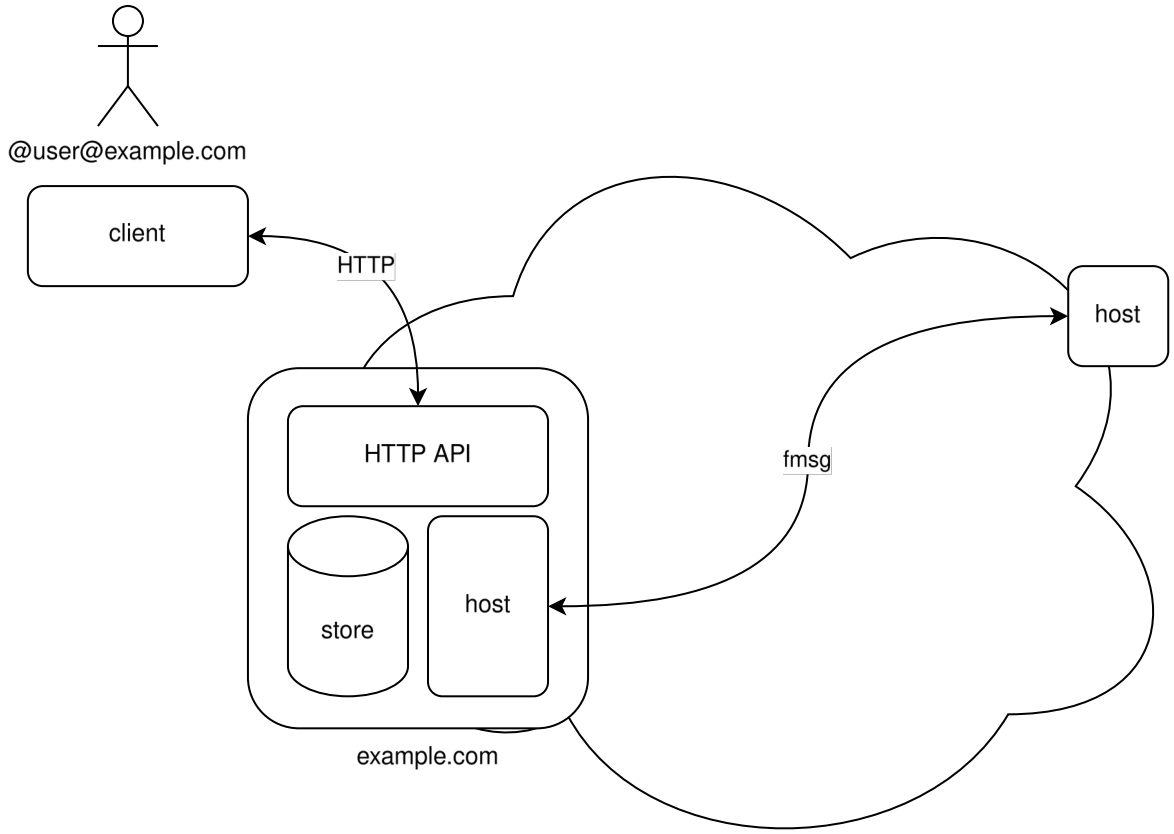


*Figure 4: fmsg architecture example*

Standards are published to facilitate interoperability between services here: https://github.com/markmnl/fmsg/blob/main/STANDARDS.md. For example the “FMSG-003 fmsg Id” standard defines an HTTP Web API for a fmsg host to lookup the identity, quotas and metadata of a user. An implementation of the FMSG-003 fmsg Id standard could then be used by a fmsg host without being tied to a specific Identity Provider.

# Real World Comparison

**Declaration of generative AI use**: for this section, Real World Comparison, the author used Claude (Anthropic; Claude Fable 5 and Claude Fable 5.1, accessed via Claude Code) to develop the benchmark harness, analysis scripts in the https://github.com/markmnl/fmsg-bench repository, and this analysis text. The author designed the test scenarios, executed all measurements on the

described infrastructure, verified the reported figures against the raw captures, and reviewed and edited the text. The author takes full responsibility for the content of the publication.

## Introduction

We compare full setups of fmsg (using the open source ecosystem of implementations: fmsgd, fmsg-webapi, fmsg-cli on three domains spanning the world) against email (using popular providers Gmail and Outlook, and a self-hosted stack, mailcow, which uses Postfix). Size of message transmitted and time taken to send are recorded over a variety of scenarios: simple one-off messages, typical message exchanges with and without attachments, forwarding, short conversations between two people, long conversations, branching conversations, and conversations amongst several people on several domains.

While care is taken to record metrics fairly and like-for-like, these tests should not be viewed as definitive benchmarks, especially the timings, which vary between software stacks, versions, operating systems and, since every leg crosses the public internet, network conditions on the day. Bear in mind also that there is a multitude of email services and software, of which only a few have been picked here. Still, a realistic feel for the size of messages in different scenarios, and for how that size grows as a conversation develops, can be gleaned from this comparison.

## Method

All email and fmsg tests are conducted from the same infrastructure: the local fmsg host and the mailcow mail host run on the same machine, and the remote parties are, for fmsg, production fmsg hosts on two other domains and, for email, Gmail and Outlook. Every test is captured as a packet trace at the local host's network interface, filtered to the remote host's addresses, so only host-to-host traffic is measured. Metrics are captured including TCP/IP and TLS framing, from first SYN to final FIN:

- **Total size** of everything transmitted between hosts in bytes (8-bit octets), both directions summed. A secondary figure, the TLS payload bytes alone, is retained in the raw data so readers can subtract TCP/IP overhead if they wish.
- **Total time** of transmission, which is not simply a function of size and bandwidth, because real implementations perform checks and controls: the built-in challenge-response of fmsg, and the various checks and queueing of a contemporary email setup.

Note that the client-to-host leg is not captured; it depends on the protocol the client uses to submit and retrieve messages (HTTP or WebSocket for fmsg-webapi, SMTP submission and IMAP for email). So we are really testing email's core host-to-host protocol, SMTP with STARTTLS on port 25, against the fmsg protocol used for host-to-host transmission, over TLS 1.3 on port 4930. Direct MX delivery is used for email in both directions; no relay or smarthost is involved.

Every scenario sends the same content on both systems: natural-language message bodies of exactly 120 bytes, and identical attachments (either incompressible random bytes generated from a fixed seed, or realistic files: a screenshot, a photo and a word-processor document). Each message

after the first is a reply to the previous one, with senders rotating when there are more than two participants. Email replies are generated through the Gmail API and Microsoft Graph with standard threading headers and the full quoted history, as real mail clients produce. fmsg replies reference their parent message by hash, as the protocol specifies.

Two behaviours of the fmsg implementation are worth stating because they are included in the figures. First, fmsgd's default challenge mode means a host receiving a message from a sender that has not yet participated in the thread opens a second connection back to the sending host to verify it; this cost is included whenever it occurs. Second, fmsgd uses TLS session resumption, so connections after the first to a given host skip the certificate exchange.

Byte counts proved stable across repetitions, so each scenario was run once per system; timings are reported as measured and should be treated as indicative. The command line tools tcpdump and tshark are used to capture and analyse the tests. The scripts, the scenario matrix, the commands used to capture and extract results, and the versions of all software are appended so the tests can be replicated. Many more scenarios could be concocted; the intent is to mimic typical real-world messaging and get a feel for how different exchanges (with and without replies, multiple recipients, attachments, forwarding) affect transmission size.

## List of Tests

Scenario IDs read `m<N>p<P>`: N messages exchanged between P participants (one sender plus P-1 recipients). Suffixes: `a10k`/`a1m` a 10 KiB / 1 MiB random attachment on the first message; `ascr`/`aimg`/`adoc` a realistic screenshot / photo / document; `x` recipients on different domains or providers; `-br` branching; `-fwd` forwarding.

*Table 1: List of tests*

| Test | Description |
|---|---|
| m1p2 | One 120-byte message to one recipient |
| m2p2 | Two messages between two participants, the second a reply to the first |
| m3p2 | Three messages between two participants, each a reply to the previous |
| m4p2 | Four messages, as above |
| m5p2 | Five messages, as above |
| m10p2 | Ten messages, as above |
| m20p2 | Twenty messages, as above |
| m100p2 | One hundred messages, as above (long-thread stress) |
| m3p2-br | Three messages between two participants; the second and third are both replies to the first (a branching conversation) |
| m1p2a10k | One message to one recipient carrying a 10 KiB incompressible attachment |
| m1p2a1m | One message to one recipient carrying a 1 MiB incompressible attachment |

| Test | Description |
|---|---|
| m1p2ascr | One message to one recipient carrying a realistic screenshot (PNG, 336 kB) |
| m1p2aimg | One message to one recipient carrying a realistic photo (JPG, 1.02 MB) |
| m1p2adoc | One message to one recipient carrying a realistic document (ODT, 377 kB) |
| m1p3 | One message to two recipients whose mailboxes are reached through one remote server (fmsg: remote domain plus the sender's own domain; email: two mailboxes at one provider) |
| m1p3x | One message to two recipients on two different remote domains (email: one at Gmail, one at Outlook) |
| m10p3x | Ten-message conversation, three participants on three domains, senders rotating |
| m10p4x | Ten-message conversation, four participants on three domains, senders rotating |
| m1p2-fwd | One message to one recipient; afterwards a third participant is brought into it (fmsg add-to; email forward)One message to one recipient; afterwards a third participant, on a domain that already holds the message, is brought into it (fmsg add-to; email forward) |
| m1p2a1m-fwd | As m1p2-fwd, but the message carries a 1 MiB attachment |

# Results

All sizes are total bytes on the wire including TCP/IP and TLS framing, both directions. Times are seconds from first SYN to last FIN.

## A single message

A single 120-byte message costs 9.6 kB on fmsg and 25.0 kB on email (Table 2). The fmsg figure is two connections of roughly equal size: the delivery itself (4.9 kB) and the challenge-response connection the receiving host opens back to the sender on first contact (4.6 kB). Each is a full TLS 1.3 handshake with certificate chains, since this is the first contact between the hosts in the run. Email's single SMTP transaction is larger than both put together: the STARTTLS upgrade and certificate exchange, the EHLO capability negotiation, and a message whose header block (tracing, authentication signatures, identifiers and MIME structure) outweighs the 120-byte body many times over.

## Conversations

Table 1: List of tests and Figure 5: Conversation growth, first 20 messages show how cost accumulates as two people exchange replies.

*Table 2: Two-party conversations results*

| Messages | fmsg bytes | email bytes | email / fmsg | fmsg time (s) | email time (s) |
|---|---|---|---|---|---|
| 1 | 9.6 kB | 25.0 kB | 2.6x | 1.4 | 3.0 |
| 2 | 14.3 kB | 35.6 kB | 2.5x | 5.6 | 21.6 |
| 3 | 19.5 kB | 55.4 kB | 2.8x | 8.3 | 22.1 |
| 4 | 24.5 kB | 70.8 kB | 2.9x | 12.5 | 33.0 |
| 5 | 29.3 kB | 92.2 kB | 3.1x | 15.1 | 39.1 |
| 10 | 54.4 kB | 186.6 kB | 3.4x | 33.1 | 77.6 |
| 20 | 103.9 kB | 425.2 kB | 4.1x | 68.9 | 125.6 |
| 100 | 501.2 kB | 4.31 MB | 8.6x | 360.7 | 886.6 |

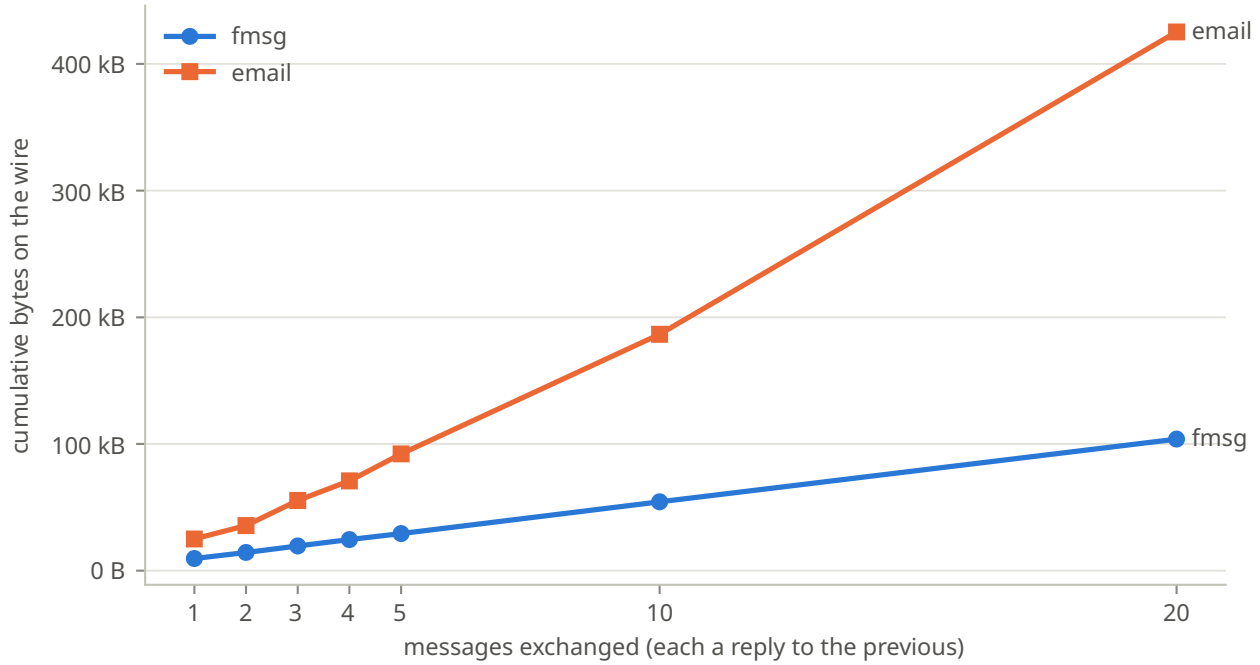


*Figure 5: Conversation growth, first 20 messages*

fmsg's line is straight. After the first message, every reply costs about 5.0 kB regardless of how deep the conversation is: a reply carries its 120-byte body and a 32-byte reference to its parent, inside a resumed TLS session, and the hosts have already verified each other for this thread so no further challenge is made. Over 100 messages the marginal cost never moves outside 4.8 to 5.2 kB per message.

Email's line curves upwards. A reply re-quotes the entire history of the conversation, so each message is larger than the last: the marginal cost is about 10 kB for the second message, around 20 kB per message by the fifth, 24 kB per message between the tenth and twentieth, and 49 kB

per message between the twentieth and hundredth. The conversation as a whole costs 2.6 times fmsg at one message, 4 times at twenty, and 8.6 times at a hundred, and the gap keeps widening with depth. With longer bodies than our 120 bytes the effect is correspondingly larger, since it is the bodies that are being re-quoted.

Quoting the previous message is a client convention rather than a requirement of SMTP. The replies here carry standard In-Reply-To and References headers and quote the history in plain text, as Gmail, Outlook and Apple Mail do by default. A client that sent only the new text would hold email's per-message cost roughly flat, at about twice fmsg's. Real clients typically also send an HTML alternative of the body and quote, along with signatures, so these figures are, if anything, conservative.

## Attachments

*Table 3: One message with an attachment*

| Attachment | fmsg bytes | over raw file | email bytes | over raw file | fmsg time (s) | email time (s) |
|---|---|---|---|---|---|---|
| 10 KiB random | 20.5 kB | +100% | 35.4 kB | +246% | 1.4 | 3.0 |
| 1 MiB random | 1.12 MB | +7% | 1.52 MB | +45% | 2.5 | 10.2 |
| Screenshot, PNG 336 kB | 361 kB | +8% | 506 kB | +51% | 2.2 | 5.2 |
| Photo, JPG 1.02 MB | 1.08 MB | +6% | 1.48 MB | +45% | 2.7 | 9.8 |
| Document, ODT 377 kB | 417 kB | +11% | 566 kB | +50% | 2.6 | 6.1 |

fmsg transmits attachments as raw binary, so for anything larger than a few kilobytes the overhead over the file itself is the fixed cost of the connection and challenge (6 to 11% here). Email base64-encodes every attachment inside MIME, which alone adds a third to the size, and the line-wrapping, boundaries and headers bring the total to 45 to 51% over the raw file, on every hop the message takes. The 10 KiB case is dominated by fixed per-connection costs on both systems and is shown for completeness.

## Multiple Recipients and Multiple Domains

*Table 4: Multi-recipient and multi-domain scenarios*

| Scenario | fmsg bytes | email bytes | email / fmsg | fmsg time (s) | email time (s) |
|---|---|---|---|---|---|
| m1p3, two recipients via one remote | 9.7 kB | 20.3 kB | 2.1x | 1.4 | 3.1 |

| Scenario | fmsg bytes | email bytes | email / fmsg | fmsg time (s) | email time (s) |
|---|---|---|---|---|---|
| server | | | | | |
| m1p3x, two recipients on two remote domains | 19.2 kB | 35.1 kB | 1.8x | 5.4 | 5.2 |
| m10p3x, ten messages, three participants, three domains | 79.1 kB | 259.8 kB | 3.3x | 99.2 | 69.6 |
| m10p4x, ten messages, four participants, three domains | 89.3 kB | 237.0 kB | 2.7x | 103.6 | 85.8 |

Both protocols fan out from the sender's own host, once per destination: one fmsg connection per recipient domain, one SMTP transaction per destination server. A second recipient at a server already being delivered to adds nothing on either system (m1p3 against m1p2). A second recipient on a second domain roughly doubles the cost on both systems (m1p3x), fmsg paying a fresh handshake and challenge with the new host, email a fresh STARTTLS session and transaction with the new provider. Across a ten-message conversation spanning three domains fmsg is about three times cheaper for the same structural reason as the two-party case: each message costs a fixed few kilobytes per domain, where each email reply re-quotes the history to every destination.

## Adding a Participant

*Table 5: Adding a participant after the message was sent*

| Scenario | fmsg total | fmsg added | email total | email added |
|---|---|---|---|---|
| m1p2-fwd, 120-byte message | 19.0 kB | +9.4 kB | 40.2 kB | +15.2 kB |
| m1p2a1m-fwd, message with 1 MiB attachment | 1.12 MB | +9.4 kB (header only) | 3.05 MB | +1.53 MB |

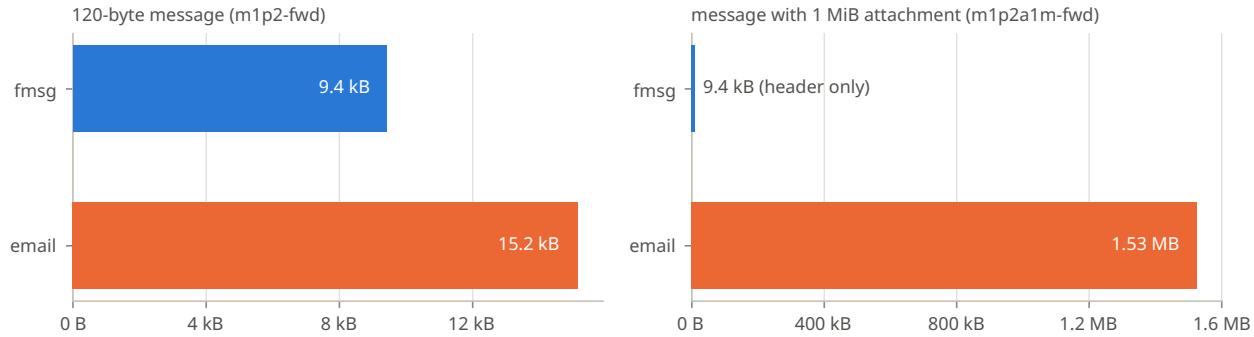


*Figure 6: Marginal cost of adding a participant (same recipient domain)*

This is where the two designs diverge most, with one qualification. An fmsg add-to sends only the message header to each participant domain; a host that already holds the message answers that it has the data, and nothing more is transmitted to it. That is the case measured here: the added participant is on the remote domain that received the original, so bringing them in costs 9.4 kB whether the message carried a byte or a megabyte. A participant on a domain that has never held the message would instead receive a full delivery, attachment included, so the general statement is that fmsg transmits a message's data to each domain once, and adding a participant thereafter costs a header per domain. Email can only forward: the whole MIME body, base64 attachment included, is retransmitted to the new recipient regardless of what their server already holds, and re-quoted on every reply-all that follows. In the measured scenario that is about 160 times fmsg's cost and 2.7 times the total bytes.

### Time

Where the two parties are a single host pair, fmsg completed every scenario faster than email: a single message in 1.4 s against 3.0 s, a five-message exchange in 15 s against 39 s, a 1 MiB attachment in 2.5 s against 10.2 s, and a hundred-message conversation in six minutes against fifteen. Much of email's time is spent between a message arriving at the provider and the reply leaving it (the second message of m2p2 alone adds about 18 s, which includes the test driver's polling of the provider's API), together with the checks a modern mail server performs before accepting a message. fmsg's time is mostly connection round trips, including the challenge on first contact. As stated in the method, all timings cross the public internet, in this case between Australia and North America, and third-party infrastructure and are indicative only; byte counts are the stable, comparable metric.

## Conclusion

Measured on the wire between real hosts on the public internet, fmsg transmits fewer bytes than email in every scenario tested. For a single short message the ratio is about 2.5 to 1, and this is with fmsg paying for a challenge-response connection that email has no equivalent of. The gap comes from the shape of the protocols rather than from tuning: fmsg carries a compact binary header and raw attachment data over a TLS 1.3 connection that is encrypted from the first byte and resumed

on subsequent contact; SMTP carries a verbose text envelope, a header block that has grown with decades of authentication and tracing extensions, and base64-encoded content.

The differences compound as messaging becomes conversational. Because an fmsg reply references its parent by a 32-byte hash, the cost of a conversation grows linearly at about 5 kB per message. Because an email reply, as mainstream clients compose it, quotes the conversation so far, its cost grows quadratically: 4 times fmsg at twenty messages, 8.6 times at a hundred, and worse for longer bodies. That growth is a consequence of client convention rather than of SMTP itself; a client that quoted nothing would still cost about twice fmsg per message, but flat. Attachments cost 45 to 50% over the file on email against 6 to 11% on fmsg, and bringing a new participant into a message with an attachment costs email the whole attachment again, where fmsg's add-to costs a header for every domain already holding the message and a single delivery to any domain that does not.

These results are from one implementation of each system on one set of infrastructure, with one repetition per scenario and short synthetic bodies, and timings in particular should be read as indicative. Even so, the direction and rough magnitude of the differences follow directly from the protocols' designs, and we would expect them to hold across other compliant implementations and providers.

# Notes on Encryption

Many IM applications claim “end-to-end encryption” (E2EE) implying only the sender and receiver of messages can see the decrypted message, despite the messages passing through the IM application’s servers. Some providers go further claiming even the meta data such as who is talking to who and when, is also encrypted on their back-end. Technically such claims are difficult to verify – we have to rely on the owner’s honesty, technical implementation and message handling. Some IM applications go a step further making their client-side code open-source thereby auditable, which combined with user controlled encryption keys should provide E2EE *if* the design is sound, the keys are safe, and there are no vulnerabilities exploited in the implementation.

Legally, a service falls under one or more legal jurisdictions – depending on where the service is operated, controlled and used – data disclosure laws may apply. This should be considered when using a 3rd party service – who owns the service and where does it operate? Users have no control with consumer IM applications, unlike email and fmsg which can either be self-hosted or at least chosen from any one provider since they are based on an open protocol – you can send an email from one provider to another, for instance one can send an email from Gmail to Yahoo, but you cannot send a message from WhatsApp to LINE, say.

The fmsg protocol itself does not encrypt messages – instead, much like email, leaves it up to user/client to encrypt message content which allows end users to make the content unreadable to even their hosts. Message headers containing recipients, time, type etc. needs to be readable by hosts. So the whole message should be encrypted in transit in a way the receiving host can decode

the headers and eavesdroppers cannot (e.g. using Transport Layer Security, TLS, typically with Public Key Infrastructure, PKI).

So the level of encryption is defined by how a host and specific address is configured. The following standards can be used

how the fmsg host is configured Indeed the use of TLS 1.3+ is mandated by the only transport and binding defined so far[4].

# Notes on Signing

DKIM signs messages proving the signer held the private key associated with the domain at the time. Used in email this enables the recipient to verify a message was signed by the claimed domain because the public key published by that domain can be used to verify the signature.

In fmsg, messages, travel direct between hosts, host-to-host – relaying is structurally prevented by the requirement the originating IP address belongs to a fmsg host resolved by the subdomain method. This combined with eventual verifiablity (or on the spot verification if the challenge response was used during the message exchange) provides similar domain-level sender verification that DKIM would, without the rigmarole of generating, safeguarding and rotating keys. In addition Domain Name System Security Extensions (DNSSEC), used when the remote supports it, provides authenticated DNS data, and TLS (the only public binding for fmsg as of writing) ensures integrity and confidentiality during transmission.

So signing to just prove provenance would be redundant at domain level. Individual addresses may still sign and publish their public keys to prove provenance for themselves – such methods are deliberately left out of the fmsg specification so different methods can be additive at the users' discretion, much like encryption of the contents. For example OpenPGP could be used to prove provenance and encrypt the body for specific address.



4 FMSG-001 TCP+TLS Transport and Binding Standard, https://github.com/markmnl/fmsg/blob/main/standards/fmsg-001-transport-and-binding.md, accessed on 5 July 2026

# Epilogue – Applicability for AI Agents

# References

# Appendix I – Links

Documents are hosted on https://fmsg.org derived from GitHub repository: https://github.com/markmnl/fmsg, including "living" documents for: standards, specification and list of implementations.

Standards are listed below and a snapshot of the full specification is appended.

## Standards

Standards follow an incrementing numbering scheme in the order they were created prefixed with "FMSG-" FMSG-001

Standards - https://github.com/markmnl/fmsg/blob/main/STANDARDS.md

Specification - https://github.com/markmnl/fmsg/blob/main/SPECIFICATION.md

**fmsgd** – An fmsg host implementation written in Go! https://github.com/markmnl/fmsgd

**fmsg-docker** – A containarised fmsg stack consisting of fmsgd, fmsg-webapi, fmsgid and backing PostgresSQL databases. https://github.com/markmnl/fmsg-docker

**fmsg-webapi** – An HTTP API for clients to send and receive messages via their fmsgd host. https://github.com/markmnl/fmsg-webapi

**fmsg-cli** – A Command Line Interface for sending and receiving messages via fmsg-webapi. https://github.com/markmnl/fmsg-cli

# Appendix II – IM Applications

*Table 6: Popular Instant Messaging Applications*

| Instant Messaging App | Approx. Monthly Active Users[5] | Company / Owner |
|---|---|---|
| WhatsApp | 3+ billion (Q1 2025) | Meta Platforms, Inc. |
| WeChat (Weixin) | 1.439 billion (Q2 2026) | Tencent Holdings Ltd. |
| Telegram | 1+ billion (Mar 2025) | Telegram FZ-LLC (founder Pavel Durov) |
| Facebook Messenger | ~1 billion (estimate) | Meta Platforms, Inc. |
| Snapchat | 971 million (Q2 2026) | Snap Inc. |
| QQ | 520 million (Q2 2026) | Tencent Holdings Ltd. |
| LINE | ~194 million (Mar 2025) | LY Corporation (SoftBank / Naver) |
| Viber | Not disclosed (est. ~100–250 million) | Rakuten Group (Rakuten Viber) |
| Zalo | 81.3 million (Q2 2026) | VNG Corporation |
| Signal | Not disclosed (est. ~70 million) | Signal Messenger LLC (Signal Foundation, nonprofit) |
| KakaoTalk | 55.3 million (Q2 2026) | Kakao Corp. |

*Table 7: Popular Group Messaging Platforms*

| Platform | Typical Use Case | Self-hostable | Company / Owner |
|---|---|---|---|
| Microsoft Teams | Enterprise workplace chat, meetings, and calling within Microsoft 365 | No | Microsoft Corporation |
| Discord | Online communities, gaming, and interest groups | No | Discord Inc. |
| Slack | Business team collaboration and workflow integrations | No | Salesforce, Inc. |
| Google Chat | Enterprise team chat within Google Workspace | No | Google LLC (Alphabet) |
| Zoom Team Chat | Workplace chat alongside Zoom meetings | No | Zoom Communications, Inc. |
| Rocket.Chat | Secure team chat and customer messaging for organizations with strict data-protection needs | Yes | Rocket.Chat Technologies Corp. (open source) |
| Mattermost | Secure collaboration for defense, government, and critical-infrastructure teams | Yes | Mattermost, Inc. (open core) |
| Zulip | Topic-threaded team chat for open-source projects, research groups, and organizations | Yes | Kandra Labs, Inc. (open source) |

5 Latest figures each company has reported as of September 2026, so the dates differ from row to row. Where no official number exists, estimated from online sources.

# Appendix III – Real World Comparison Tests

Refer to the https://github.com/markmnl/fmsg-bench GitHub repository for actual code and all results, in addition to the summarised results earlier in this document. Below are pseudo code outlines of that code.

## A.1 Orchestrator (one run per system)

```
for each scenario in scenarios.tsv selected for this system:
  for rep in 1..REPS:                        # REPS = 1 for fmsg and email
    if results.csv already has (system, scenario, rep): skip

    start tcpdump on the local host's interface
        fmsg:  filter "tcp port 4930",  remote = resolved IPs of remote hosts
        email: filter "tcp port 25",    remote = Google + Microsoft SMTP
netblocks
    wait until the capture is live

    run_scenario(scenario)                   # A.2
    sleep 2 seconds                          # let final FINs land
    stop tcpdump

    extract(pcap) -> one row in results.csv  # A.4
    email only: sleep 10 s so the MTA's connection cache expires
                and the next rep negotiates its own connection
```

## A.2 Scenario (shared by both systems)

Scenario `m<N>p<P>` with optional attachment and modifier `x`, `br` or `fwd`. Participant 1 is the local party; participants 2..P are remote.

```
run_scenario(N messages, P participants, attachment, modifier):
  for n in 1..N:
    sender     = 1                     if n == 1
               = 2                     if modifier == br     # every reply from
participant 2
               = ((n-1) mod P) + 1     otherwise             # 2-party
alternates, else round-robin
    recipients = all participants except sender
    body       = 120 bytes of natural-language text, prefixed with a
                 unique tag identifying (scenario, rep, n)
    parent     = none                  if n == 1
               = message 1             if modifier == br     # branch: reply to
the first message
               = message n-1           otherwise             # chain: reply to
the previous one
    attach     = attachment file       if n == 1 else none

    send(sender, recipients, body, parent, attach)            # A.3
    wait until the sender's host reports delivery to every recipient
    wait until the next sender's inbox holds the message with this tag

  if modifier == fwd:                  # bring participant P+1 into message 1
    fmsg:  participant 1 issues add-to(message 1, participant P+1)
           wait for add-to delivery, then for message 1 in P+1's inbox
```

```
    email: participant 1 sends "Fwd:" of message 1, attachment re-attached
           in full, to a new mailbox; wait for it to arrive
```

## A.3 Send and wait primitives

```
fmsg   send:  fmsg-cli (via fmsg-webapi on the sender's host)
              draft -> set recipients -> attach file -> send, with
              parent id (pid) when replying. The host then delivers to
              each recipient domain over TLS on port 4930; on first
              contact for a thread the receiving host opens a
              challenge-response connection back to the sender.
       wait:  poll the sender's "sent" list every 0.5 s until every
              recipient shows time_delivered; poll the recipient's inbox
              until a message whose body starts with the tag appears.

email  send:  local party    submits via mailcow (SMTP submission over an
                             SSH tunnel; outside the capture) with
                             In-Reply-To and References set and the
                             previous body quoted below the new text.
              Gmail party    replies through the Gmail API, same headers,
                             same quoting; reply-all when P > 2.
              Outlook party  replies through Microsoft Graph replyAll.
              mailcow then delivers direct to each provider's MX on port
              25 (STARTTLS); providers deliver back to mailcow on port 25.
       wait:  poll the Gmail API / IMAP on mailcow (outside the capture)
              until the message with the tag has arrived.
```

## A.4 Extraction (per capture)

```
extract(pcap):
  packets  = all TCP packets on the scenario's port whose remote
             endpoint is within the configured remote addresses
  streams  = group packets by TCP connection
  bytes_ab = sum of frame bytes sent by the local side, over all streams
  bytes_ba = sum of frame bytes sent by the remote side
  bytes_total     = bytes_ab + bytes_ba          # includes TCP/IP + TLS framing
  tls_bytes_total = sum of TCP payload bytes     # TLS records only
  t_first_syn = earliest SYN in any stream
  t_last_fin  = latest FIN in any stream
  duration_s  = t_last_fin - t_first_syn
  write (system, scenario, rep, number of streams, bytes_ab, bytes_ba,
         bytes_total, tls_bytes_total, duration_s) to results.csv
  write per-stream detail to <pcap>.streams.json
```

Packets are read with tshark where available, otherwise with a built-in pcap parser that produces identical rows. Reported figures are the per-scenario median over repetitions. Software versions, payload checksums and the exact commands used are recorded in `results/versions.txt` and the repository README.

# Appendix IV – Specification

This full specification is hosted at https://fmsg.org. Appended is the v0.6.0 from 11 September 2026. Hyperlinks have been left in and refer to original source in GitHub: https://github.com/markmnl/fmsg.

## Specification Contents

# Terminology

*"fmsg"* is the name given to the protocol and message definitions described in this document. The name "fmsg" is neither an abbreviation nor acronym, instead is thought of as "f-message". The "f" is inspired from popular programming languages such as C's `printf` where the "f" stands for "formatted", "msg" is a common shortening of "message" conveying the meaning while keeping the whole name succinct; "fmsg".

This document occasionally uses RFC style normative language (e.g., "MUST", "SHOULD", "MAY") to better describe protocol behaviour and requirements. However, this specification is not currently an official RFC and does not claim standards-track status. The use of such terminology is intended to improve clarity in defining the protocol. Future evolution of this work may align with the RFC process, at which point an updated specification would use these terms in accordance with such established conventions.

## Terms

*"address"* an fmsg address in the form `@user@example.com`, see: Address.

*"case-insensitive"* byte-wise equality comparison after applying Unicode default case folding (locale-independent) to both UTF-8 strings.

*"client"* the end participant/application that sends and receives messages via their *host*.

*"DNS"* is for the Domain Name System.

*"host"* is an fmsg implementation which can send and receive fmsg messages to and from other hosts following the definitions and protocol of this specification.

*"message"* refers to an entire message described in Message definition.

*"message hash"* the SHA-256 digest of a message computed per Computing Message Hash.

*"message header"* refers to the fields up to and including the attachment headers field in a message.

*"message header hash"* the SHA-256 digest of a message header.

*"participants"* all recipients plus *from*, plus *add to from* (if exists)

*"recipient"* an address in a message's *to* or *add to* fields

*"recipients"* the addresses in a message's *to* and *add to* fields

*"sender"* the address in a message's *from* field when *has add to* not set; otherwise the address in the *add to from* field.

*"terminal message"* a message with the *terminal* flag bit set. No message may reference it via *pid*, see Notes on Terminal Messages.

*"thread"* is a linked hierarchy of messages where messages relate to previous messages using the *pid* field

*"UTF-8"* is for the unicode standard: Unicode Transformation Format – 8-bit.

## Overview

Before diving into the technical details, the following principles outline how fmsg works at a high level.

**Messages are immutable.** Messages can never be changed once sent.

**Messages form threads.** Every reply references the previous message it is responding to via a cryptographic hash. This creates a linked chain of messages – a thread – where each message's parentage is verifiable by all participants. The first message in a thread has no parent reference and instead carries a topic.

**One message at a time.** A host sends a single message per connection to a receiving host. The message is either rejected outright for all recipients (e.g. the message is malformed or too large) or accepted and rejected on a per-recipient basis (e.g. a recipient's message store is full while another's accepts the message).

**Binary and compact.** Messages are encoded in a binary format with explicitly sized fields. There are no null terminators or delimiters – every field's length is known, keeping messages compact and resistant to common parsing vulnerabilities.

**Only participants can reply.** To reply to a message a sender must have been a participant of that message. This is enforced structurally: the hash used to reference a parent depends on whether the sender was in the original recipient list, the original sender or was added later, and the Receiving Host verifies this linkage.

**Rejection tells you why.** When a message is rejected the Receiving Host includes a reason code. This allows the Sending Host to determine whether re-sending is worthwhile (e.g. the recipient was temporarily full), pointless (e.g. the message is a duplicate), or indicative of a problem that needs attention (e.g. the message was deemed invalid).

**Sender verification is built in.** A Receiving Host verifies that the sending host's IP address is authorised by the sender's domain via DNS. An optional challenge mechanism provides additional assurance by requiring the sender to prove knowledge of the message content while it is being transmitted.

# Definition

## Message Types

Four message types are defined by fmsg: MESSAGE, CHALLENGE, CHALLENGE RESPONSE and "REJECT or ACCEPT RESPONSE". These structures are aggregates of Data Types and are described in the Definition section.

## Data Types

Throughout this document the following data types are used. All types are always encoded little-endian.

| **name** | **description** |
|---|---|
| uint8 | 8 bit wide unsigned integer with a value in the set 0 to 255 |
| uint16 | 16 bit wide unsigned integer with a value in the set 0 to 65535 |
| uint32 | 32 bit wide unsigned integer with a value in the set 0 to 4294967295 |
| bit | single bit 0 or 1 within one of the uint types, the 0 based index of which is defined alongside in this document |
| float64 | 64 bit wide number in the set of all IEEE-754 64-bit floating-point numbers |
| byte | a uint8 |
| byte array | sequence of uint8 values the length of which is defined alongside in this document |
| bytes | a sequence of bytes |
| string | sequence of characters the length and encoding (e.g. US-ASCII, UTF-8...) of which is defined alongside in this document |

String lengths are always explicitly defined and null terminating characters are not used. This is a design decision because it prevents a class of buffer over-run bugs (search "Heartbleed bug"), simplifies message size calculation, and, inherently limits the length of strings while adding no extra data than a null terminating character would (since all strings lengths here are defined by one uint8).

## Message

In programmer friendly JSON a message could look like (once decoded from the binary format defined below):

```
{
    "version": 1,
    "important": false,
    "noreply": false,
    "terminal": false,
    "pid": null,
    "from": "@user@example.com",
    "to": [
```

```
        "@世界@example.com",
        "@chris@example.edu"
    ],
    "time": 1654503265.679954,
    "topic": "Hello fmsg!",
    "type": "text/plain;charset=UTF-8",
    "size": 45,
    "data": "The quick brown fox jumps over the lazy dog.",
    "attachments": [
        {
            "type": "application/pdf",
            "filename": "doc.pdf",
            "size": 1024
        }
    ]
}
```

On the wire messages are encoded thus:

| **field** | **type** | **description** |
|---|---|---|
| version | uint8 | Values 1 through 127 is the fmsg version number, values 129 through 255 means this message is a CHALLENGE defined below. Values 0 and 128 are unused. |
| flags | uint8 | Bit field. See flags for each bit's meaning. |
| [pid] | byte array | 32 byte SHA-256 digest referencing the parent message hash. Only present if flags has pid bit set. |
| from | fmsg address | Sender's address. |
| to | uint8 + list of fmsg addresses | Recipient addresses. Prefixed by uint8 count, addresses MUST be distinct (case-insensitive) of which there MUST be at least one. |
| [add to from] | fmsg address | Address adding recipients. MUST exist if flags has *add to* bit set, otherwise MUST NOT exist. |
| [add to] | uint8 + list of fmsg addresses | Additional recipient addresses. MUST exist if flags has *add to* bit set, otherwise MUST NOT exist. Prefixed by uint8 count, addresses MUST be distinct (case-insensitive) of which there MUST be at least one. |
| time | float64 | POSIX epoch time message was ready for sending by host sending the message. |
| [topic] | uint8 + UTF-8 string | UTF-8 free text title of the first message in a thread. Only present if *pid* is not set. Prefixed by uint8 size which may be 0. |
| type | uint8 + [US-ASCII string] | Either a common type, see Common Media Types, or a US-ASCII encoded Media Type: RFC 6838. |
| size | uint32 | Size of data in bytes, 0 or greater. Data may be compressed if *zlib*- |

| field | type | description |
|---|---|---|
| | | *deflate* header bit is set, *size* is number of bytes in message transmitted over the wire i.e. after any compression applied. |
| [expanded size] | uint32 | Size of data in bytes after decompression. Present if and only if the *zlib-deflate* flag bit is set. |
| attachment headers | uint8 + [list of attachment headers] | See attachment header definition. Prefixed by uint8 count of attachments of which there may be 0. |
| data | byte array | The message body of type defined in type field and size in the size field |
| [attachments data] | byte array(s) | Sequential sequence of octet boundaries of which are defined by attachment headers size(s), if any. |

## Notes on Message Definition

- Square brackets "[ ]" indicate fields or part thereof may not exist on a message. Where the brackets surround the name, e.g. *pid*, the whole field may not be present (which in the case of pid is only valid if the message is the first in a thread). Where they surround part of the type, that part may not be present, e.g. list of attachment headers will not be present if uint8 prefix is 0.
- *topic* only exists on the first message in a thread, i.e. on a message with no *pid*. This makes *topic* immutable because it cannot be changed by subsequent replies. (Presentations of message threads MAY use a local mutable field for display purposes).
- It is not possible to accept a message *from* an address that wasn't a participant in the message referenced by *pid* following the Protocol Steps.
- It is not possible to accept a message whose *pid* references a message with the *terminal* flag bit set, see Notes on Terminal Messages.

## Notes on Adding Recipients

Adding recipients is achieved by sending a whole new distinct message, that is an exact duplicate of the message to which recipients are being added, except:

- The *has add to* flag bit is set
- The *has pid* flag bit is set and *pid* references the message which recipients are being added to (replacing any *pid* the original message had).
- *topic* is omitted if the original message had one, because *pid* is now present and *topic* only exists on a message without *pid*.

- *add to from* exists and is the address of the participant in the previous message adding the additional recipients, i.e. the sender.
- *add to* exists and is addresses of the new recipients being added, which MAY include an address already in *to* – re-serving an original recipient.
- *time* is the POSIX epoch time of this new message with added recipients was ready for sending.

An add-to message MUST be sent to every participant domain, not only the unique domains of the message's recipients, so that all participants of the message being added to – including the original sender, when not themselves the *add to from* – learn of the added recipients. This is required because a subsequent reply may reference the add-to message via *pid*, and a host can only accept a reply whose parent it holds. See 4. Sending a Message.

Add-to batches do not chain: recipients are always added to the original message; an add-to message's *pid* MUST NOT reference another add-to message. A message therefore has 0 or more add-to batches, each independently referencing it, and each batch forms its own sibling branch under the original message – the thread evolves as a tree rooted at the original.

A recipient added in a batch who was not already in the original's *to* is a participant of that add-to batch message only, not of the original message. A reply from such a recipient MUST therefore reference the add-to batch message via *pid* – referencing the original message directly would fail the participant rule in Notes on Message Definition, because they are not a participant of the original. Replies from added recipients thus extend the batch's branch of the thread. An address appearing in both *to* and *add to* was already a participant of the original and may reply on either branch.

A batch is identified by its message hash (see Computing Message Hash), which covers *time*. Re-issuing an add-to with the same *add to* addresses at a new *time* is a new, distinct batch – a new sibling branch – not a duplicate.

An add-to message MUST NOT reference a terminal message, see Notes on Terminal Messages.

## Notes on Terminal Messages

A message with the *terminal* flag bit set is a leaf of its thread: nobody can reply to it and recipients cannot be added to it, because no subsequent message may reference it via *pid*. Receiving Hosts enforce this per Protocol Steps and Sending Hosts MUST NOT transmit a message that violates it, so every host holding a thread agrees on its shape – a terminal message never gains children on one host that another host rejected.

*terminal* differs from *no reply*. *no reply* is advisory: the sender states it will discard replies, but a reply is still a valid message that hosts accept and other participants see. *terminal* is enforced: a reply is rejected by every host before it is stored. The two MAY be combined.

*terminal* MAY be set on any message, including the first in a thread, and applies only to the message it is set on; it does not affect the message's parent or siblings. Because an add-to message

is an exact duplicate of the original apart from the fields listed in Notes on Adding Recipients, an add-to message carrying the *terminal* flag bit necessarily references a terminal original and is therefore invalid.

This flag exists so standards can define kinds of message that MUST NOT be built upon – for example a reaction to a message – without every host needing to understand the standard: a host enforces *terminal* without knowing why it was set. A host that predates this flag treats it as a reserved bit and rejects the message (see Flags), which is preferable to accepting it as an ordinary reply that could then be replied to.

## Notes on Time

Only one time field is present on a message and this time is stamped by the sending host when it acquired the message to be sent. (Implementations MAY associate additional timestamps with messages, such as the time message was delivered).

fmsg includes some time checking and controls, rejecting messages too far in future or past compared to current time of the receiver, and, checking replies cannot claim to be sent before their parent (See Reject or Accept Response). Of course this all relies on accuracy of clocks being used, so some leniency is granted determined by the receiving host. Bearing in mind a host may not be reachable for some time so greater leniency SHOULD be given to messages from the past. Since the time field is stamped by the sending host – one need only concern themselves that their clock is accurate.

## Flags

*Table 8: Flags bits assignment*

| bit index | name | description |
|---|---|---|
| 0 | has pid | Set if this message is in reply to another and pid field is present. |
| 1 | has add to | Set if "add to" field is included i.e. this message is copy of an existing message with recipients added |
| 2 | common type | Indicates the type field is just a uint8 value and Media Type can be looked up per Common Media Types |
| 3 | important | Sender indicates this message is IMPORTANT! |
| 4 | no reply | Sender indicates any reply will be discarded. |
| 5 | zlib-deflate | Message data is compressed using the zlib structure (defined in RFC 1950), with the deflate compression algorithm (defined in RFC 1951). |
| 6 | terminal | This message is a leaf: no subsequent message may reference it via *pid*. A Receiving Host rejects any reply to, or add-to batch of, a terminal message with REJECT code 1 (invalid). See Notes on Terminal Messages. |
| 7 | reserved | Unused, reserved for future use. MUST be 0; a Receiving Host MUST |

| bit index | name | description |
|---|---|---|
| | | respond REJECT code 1 (invalid) if set. |

### *Common Media Types*

If the common type flag bit is set, the type field consists of one uint8 value which maps to a complete Media Type string, including any parameters, exactly as listed in the table below. If the common type bit is not set, the first uint8 is the length of the subsequent US-ASCII encoded complete Media Type string, including any parameters, per RFC 6838. If the common type flag bit is set and the value has no mapping in the table below – the message is invalid and should be rejected with REJECT code 1 (invalid).

For reference the current IANA list of Media Types is located here.

*Table 9: Common Media Types mapping*

| number | Media Type |
|---|---|
| 1 | application/epub+zip |
| 2 | application/gzip |
| 3 | application/json |
| 4 | application/msword |
| 5 | application/octet-stream |
| 6 | application/pdf |
| 7 | application/rtf |
| 8 | application/vnd.amazon.ebook |
| 9 | application/vnd.ms-excel |
| 10 | application/vnd.ms-powerpoint |
| 11 | application/vnd.oasis.opendocument.presentation |
| 12 | application/vnd.oasis.opendocument.spreadsheet |
| 13 | application/vnd.oasis.opendocument.text |
| 14 | application/vnd.openxmlformats-officedocument.presentationml.presentation |
| 15 | application/vnd.openxmlformats-officedocument.spreadsheetml.sheet |
| 16 | application/vnd.openxmlformats-officedocument.wordprocessingml.document |
| 17 | application/x-tar |
| 18 | application/xhtml+xml |
| 19 | application/xml |
| 20 | application/zip |

| number | Media Type |
|---|---|
| 21 | audio/aac |
| 22 | audio/midi |
| 23 | audio/mpeg |
| 24 | audio/ogg |
| 25 | audio/opus |
| 26 | audio/vnd.wave |
| 27 | audio/webm |
| 28 | font/otf |
| 29 | font/ttf |
| 30 | font/woff |
| 31 | font/woff2 |
| 32 | image/apng |
| 33 | image/avif |
| 34 | image/bmp |
| 35 | image/gif |
| 36 | image/heic |
| 37 | image/jpeg |
| 38 | image/png |
| 39 | image/svg+xml |
| 40 | image/tiff |
| 41 | image/webp |
| 42 | model/3mf |
| 43 | model/gltf-binary |
| 44 | model/obj |
| 45 | model/step |
| 46 | model/stl |
| 47 | model/vnd.usdz+zip |
| 48 | text/calendar |
| 49 | text/css |
| 50 | text/csv |
| 51 | text/html |
| 52 | text/javascript |

| number | Media Type |
|---|---|
| 53 | text/markdown |
| 54 | text/plain;charset=US-ASCII |
| 55 | text/plain;charset=UTF-16 |
| 56 | text/plain;charset=UTF-8 |
| 57 | text/vcard |
| 58 | video/H264 |
| 59 | video/H265 |
| 60 | video/H266 |
| 61 | video/ogg |
| 62 | video/VP8 |
| 63 | video/VP9 |
| 64 | video/webm |

## Attachment

Each attachment header consists of flags, type, filename, size, and conditional expanded size fields:

| name | type | comment |
|---|---|---|
| flags | uint8 | Bit field. See attachment flags below. |
| type | uint8 + [ASCII string] | Either a common type, see Common Media Types, or a US-ASCII encoded Media Type: RFC 6838. Encoding determined by this attachment's own *common type* flag. |
| filename | string | UTF-8 prefixed by uint8 size. |
| size | uint32 | Size of attachment data in bytes, after compression applied if corresponding zlib-deflate attachment flag is set. uint32 is the max theoretical size, but hosts can/should accept less. |
| [expanded size] | uint32 | Size of attachment data in bytes after decompression. Present if and only if this attachment's *zlib-deflate* flag bit is set. |

### *Attachment Flags*

| bit index | name | description |
|---|---|---|
| 0 | common type | Indicates this attachment's type field is just a uint8 value and Media Type can be looked up per Common Media Types. |
| 1 | zlib-deflate | Attachment data is compressed using the zlib structure (defined in RFC 1950), with the deflate compression algorithm (defined in RFC 1951). |

| bit index | name | description |
|---|---|---|
| 2 – 7 | reserved | Unused, reserved for future use. MUST be 0; a Receiving Host MUST respond REJECT code 1 (invalid) if any is set. |

filename MUST be:

- UTF-8
- any letter in any language, or any numeric characters (`\p{L}` and `\p{N}` Unicode Standard Annex #44 and #18)
- the hyphen "-", underscore "_", single space " " or dot "." characters non-consecutively and not at beginning or end
- unique amongst attachments, case-insensitive
- less than 256 bytes length

Attachment data

| name | type | comment |
|---|---|---|
| data | byte array | Sequence of octets located after all attachment headers, boundaries of each attachment are defined by corresponding on-wire size in attachment header(s) |

## Address

@alice@example.com

Domain part is the domain name RFC-1035 owning the address. Recipient part identifies the recipient known to hosts for the domain. A leading "@" character is prepended to distinguish from email addresses. The secondary "@" separates recipient and domain name as per norm.

Recipient part is a string of characters which MUST be:

- UTF-8
- any letter in any language, or any numeric characters (`\p{L}` and `\p{N}` Unicode Standard Annex #44 and #18)
- the hyphen "-", underscore "_" or dot "." characters non-consecutively and not at beginning or end
- unique on host using case-insensitive comparison
- less than 256 bytes length when combined with domain name and @ characters

A whole address is encoded UTF-8 prepended with size:

| name | type | comment |
|---|---|---|
| address | uint8 + string | UTF-8 encoded string prefixed with uint8 size |

## Challenge

| name | type | comment |
|---|---|---|
| version | uint8 | Challenge version, decrements from 255 corresponding to fmsg protocol version, 255 is CHALLENGE for fmsg protocol version 1, 254 would be CHALLENGE for fmsg protocol version 2 etc. |
| header hash | 32 bytes | SHA-256 digest of message header being sent/received up to and including attachment headers. |

## Challenge Response

A challenge response is the next 32 bytes received in reply to challenge request – the existence of which indicates the sender accepted the challenge. This SHA-256 hash MUST be kept to ensure the complete message (including attachments) once downloaded matches per Verifying Message Stored.

| name | type | comment |
|---|---|---|
| msg hash | 32 byte array | SHA-256 digest of the message computed per Computing Message Hash. |

## Reject or Accept Response

A code less than 11 indicates rejection for all recipients belonging to the Receiving Host's domain and will be the only value.

Code 11 accepts an add-to message and completes the exchange: the Receiving Host already holds the rest of the message, so there is no need to send it again, and it hosts none of the *add to* recipients, so no per-recipient codes follow. (The host holds the message data because its domain was a participant of the original message the add-to references via *pid* – see 4. Sending a Message.) A host that does host *add to* recipients responds 65 (skip data) instead.

Code 64 indicates to the sender the receiving host has found the message header acceptable and transmission of the message data and any attachment data should proceed.

Code 65 indicates to the sender the receiving host already has the message (and any attachments) data, but *add to* recipients belonging to the Receiving Host still need to be processed, so skip sending data and proceed to the read per-recipient response step.

Other codes 100 and above are per recipient in the same order as recipients for the Receiving Host's domain.

| name | type | comment |
|---|---|---|
| codes | byte array | a single or sequence of uint8 codes |

| code | name | description |
|---|---|---|
| 1 | invalid | the message header fails verification checks, i.e. not in spec, including a *pid* referencing a terminal message |
| 3 | undisclosed | no reason is given |
| 4 | too big | total size or expanded size exceeds host's maximum permitted size of messages |
| 5 | insufficient resources | such as disk space to store the message |
| 6 | parent not found | parent referenced by pid SHA-256 not found and is required |
| 7 | too old | timestamp is too far in the past for this host to accept |
| 8 | future time | timestamp is too far in the future for this host to accept |
| 9 | time travel | timestamp is before parent timestamp |
| 10 | duplicate | message has already been received by this host |
| 11 | accept add to | additional recipients received, discontinue |
| | | |
| 64 | continue | header received, continue message transmission |
| 65 | skip data | header received, skip sending message and attachment data |
| | | |
| 100 | user unknown | the recipient message is addressed to is unknown by this host |
| 101 | user full | insufficient resources for specific recipient |
| 102 | user not accepting | user is known but not accepting new messages at this time |
| 103 | user duplicate | message has already been received for this recipient |
| 105 | user undisclosed | no reason is given for not accepting messages to addressed recipient |
| | | |
| 200 | accept | message received for recipient |

# Protocol

A message is sent from the sender's host to each unique recipient host (i.e. each domain only once even if multiple recipients at the same domain). Sending a message either succeeds or fails per recipient of the host for the domain being sent to. During the sending from one host to another several steps are performed depicted in the below diagram. Two connection-orientated, reliable, in-order and duplex transports are required to perform the full flow. Transmission Control Protocol (TCP) is an obvious choice, on top of which Transport Layer Security (TLS) may meet your encryption needs. This specification is independent of transport mechanisms which are instead defined as standards such as: FMSG-001 TCP+TLS Transport and Binding Standard.

<Figure already included above Figure 3: Protocol flow diagram>

## Protocol Steps

The below steps are described following the example `@A@example.com` is sending a message to `@B@example.edu` for clarity. There could be more recipients on the same or different domains in a message being sent, the steps include how to handle those recipients in-situ; otherwise each recipient host performs the same steps without regards to other recipient hosts. If at any step TERMINATE is reached the message exchange is aborted. If at any step completing the message exchange is reached no further steps are performed.

*NOTE* Responding with the applicable REJECT code helps sending hosts and their clients to know when re-sending may be worthwhile (e.g. "user full"), re-sending would not be worthwhile (e.g. "duplicate") or there is an issue with either side that warrants further investigation (e.g. "invalid" suggests something wrong with the implementation, "future time" is likely due to one or both hosts' clocks being incorrect). Especially helpful to a sender is REJECT code 6 (parent not found) – the Sending Host can then indicate to its clients the Receiving Host does not hold the parent message. The client could then add the recipient as an additional recipient to continue the thread from that message (because *add to* recipients do not require the parent message to exist); or re-send previous message(s) in the thread.

The following variables corresponding to host defined configuration are used in the below steps.

| Variable | Example Value | Description |
|---|---|---|
| MAX_SIZE | 1048576 | Maximum allowed message data and attachment data size in bytes as transmitted over the wire |
| MAX_EXPANDED_SIZE | 1048576 | Maximum allowed message data and attachment data size in bytes after decompression. Implementations SHOULD normally set this to MAX_SIZE |
| MAX_MESSAGE_AGE | 700000 | Maximum age since message *time* field for message to be accepted (seconds) |
| MAX_TIME_SKEW | 20 | Maximum tolerance for message *time* field to be ahead of current time (seconds) |

### *1. Connection and Header Exchange*

1. The Sending Host (Host A) initiates a connection (Connection 1) to the first Receiving Host (Host B) authorised IP address determined by Domain Resolution.
    1. If the first IP address is unresponsive within an implementation-defined timeout, and multiple IP addresses were returned during domain resolution, Host A SHOULD attempt to connect to each address in the order provided, one at a time, until a responsive host is reached. Preserving the order of IP addresses allows the Sending Host to benefit from any load balancing, latency optimisation, or geographic routing applied during domain resolution.
    2. If no responsive Receiving Host is found, Host A SHOULD retry delivery after an implementation-defined delay. Implementations SHOULD apply a back-off strategy (e.g. exponential back-off) to subsequent retry attempts.
    3. Host A SHOULD continue retrying delivery until a maximum delivery window has elapsed, after which the message MAY be considered undeliverable. Implementations MAY provide a mechanism for clients or operators to manually trigger or influence retry behaviour.
2. Once connection is established Host A starts transmitting the message to Host B.
3. Host B downloads the first byte
    1. If the value is less than 128 and a supported fmsg version, continue.
    2. If the value is greater than 128 and 256 minus the value is a supported fmsg version – this is an incoming CHALLENGE and should be processed per Handling a Challenge.
    3. Otherwise Host B MUST TERMINATE the connection (unsupported version – we don't know how to respond).
4. Host B downloads the remaining message header and parses the fields. If parsing fails because types cannot be decoded, Receiving Host MUST TERMINATE the message exchange.
    1. The following conditions MUST be met otherwise Host B MUST respond REJECT code 1 (invalid) and close the connection completing the message exchange:
        1. There must be at least one address in *to*.
        2. All recipients in *to* are distinct using case-insensitive comparison.
        3. If the *has add to* flag bit is set:
            1. *add to from* MUST exist and *add to from* MUST also be in *from* or *to*.
            2. *add to* MUST have at least one address and all addresses in *add to* MUST be distinct using case-insensitive comparison. *NOTE I add to* requires *add to from* to be a participant of the original message, so

recipients only in *add to* cannot add recipients. *NOTE II* An address in *to* MAY also appear in *add to*. This allows an original recipient who no longer has the message to be served it again as an additional recipient. Addresses need only be distinct within each list; an address in both is a recipient of each, and receives one response code for its *to* entry and one for its *add to* entry, per Continue, Per-Recipient Response and Disposition – so both hosts always agree on how many codes the response stream contains.

4. If the *has add to* flag bit is not set, there must be at least one recipient in *to* for Host B (example.edu domain). If the *has add to* flag bit is set, there must be at least one participant (*from*, *to*, *add to from* or *add to*) for Host B.
5. *type* number when *common type* Flag is set, exists in Common Media Type mapping.
6. Each attachment *type* number, when that attachment's *common type* flag is set, exists in Common Media Type mapping.
7. The message *expanded size* field MUST exist if and only if the message *zlib-deflate* flag bit is set.
8. Each attachment *expanded size* field MUST exist if and only if that attachment's *zlib-deflate* flag bit is set.
9. If the *has add to* flag bit is set, the *terminal* flag bit MUST NOT be set. An add-to message duplicates the original's flags, so *terminal* here means the original is terminal and cannot be referenced, see Notes on Terminal Messages.
10. No reserved flag bit is set: message flags bit 7, and each attachment's flags bits 2 through 7, MUST be 0.

2. Receiving Host B MUST perform a DNS lookup on the fmsg subdomain of the senders domain to verify that the IP address of the incoming connection is in those authorised by the sending domain. If the incoming IP address is not in the authorised set, Host B MUST TERMINATE the message exchange. See Domain Resolution for more.
    1. If the *has add to* flag bit set the sender's domain is the domain part of the *add to from* address.
    2. Otherwise, the sender's domain is the domain part of the *from* address.
3. If *size* plus all *attachment size* is greater than MAX_SIZE, Host B MUST respond REJECT code 4 (too big) then close the connection completing the message exchange.

4. If the total expanded size is greater than MAX_EXPANDED_SIZE, Host B MUST respond REJECT code 4 (too big) then close the connection completing the message exchange. The total expanded size is the message *expanded size* when present, otherwise message *size*, plus each attachment's *expanded size* when present, otherwise that attachment's *size*.

5. Current POSIX epoch time minus message *time* gives DELTA – representing seconds since message sent (to senders host for sending on).
    1. If DELTA is greater than MAX_MESSAGE_AGE, Host B MUST respond REJECT code 7 (too old) then close the connection completing the message exchange.

    2. If DELTA is negative and absolute DELTA is greater than MAX_TIME_SKEW, Host B MUST respond REJECT code 8 (future time) then close the connection completing the message exchange.

6. The *pid* field requirements depends on whether this message includes additional recipients determined by the *has add to* flag.
    1. If neither *pid* nor *add to* exist, the message is the first in a thread, continue.

    2. If *pid* exists and *add to* does not.
        1. The message *pid* refers to MUST be verified to be stored already on Host B per Verifying Message Stored; otherwise Host B MUST respond with REJECT code 6 (parent not found) completing the message exchange.

        2. The stored message for *pid*'s *time* minus MAX_TIME_SKEW MUST be before *time* on the incoming message header; otherwise Host B MUST respond with REJECT code 9 (time travel) completing the message exchange.

        3. *from* MUST have been a participant in the stored message referred to by *pid*; otherwise Host B MUST respond with REJECT code 1 (invalid) completing the message exchange.

        4. The stored message referred to by *pid* MUST NOT have the *terminal* flag bit set; otherwise Host B MUST respond with REJECT code 1 (invalid) completing the message exchange. *NOTE* Verifying Message Stored checks the host has the parent message, not that every recipient still has it in their message store. Implementations MAY consider restoring the parent message to a recipient's message store if that *recipient* no longer has the message, so that the incoming reply has proper thread context for all recipients.

    3. Else *add to* exists;

1. *pid* field MUST exist too, otherwise Host B MUST respond REJECT code 1 (invalid) and close the connection completing the message exchange.
2. Verifying Message Stored is performed for message referred to by *pid*;
3. If original message referred to by *pid* is verified to be stored AND;
    1. The stored message MUST NOT have the *terminal* flag bit set; otherwise Host B MUST respond with REJECT code 1 (invalid) completing the message exchange.
    2. The stored message for *pid*'s *time* minus MAX_TIME_SKEW MUST be before *time* on the incoming message header; otherwise Host B MUST respond with REJECT code 9 (time travel).
4. Otherwise (original message has not been found, possible because Host B was never a participant of the message, or the message referenced by *pid* is no longer held):
    1. If at least one recipient in *to* or *add to* belongs to Host B, the message is treated as a full message delivery.
    2. Otherwise Host B hosts only non-recipient participants of the message; Host B MUST respond REJECT code 6 (parent not found) then close the connection completing the message exchange.

## *2. The Automatic Challenge*

A recipient fmsg host is responsible for challenging a sender for detail of the message being sent, while it is being sent, before deciding whether to continue downloading the message. A sender MUST be listening and respond to such a challenge on the same IP address as the outgoing message.

For example, a host MAY implement different challenge modes of operation such as:

1. NEVER, recipient host never sends a CHALLENGE.
2. ALWAYS, recipient host will always send a CHALLENGE during the message exchange.
3. HAS_NOT_PARTICIPATED, recipient host will send a CHALLENGE when *pid* does not exist or none of the linked messages in the thread following each message's *pid* are from a recipient on Host B's domain.
4. DIFFERENT_DOMAIN, recipient host will always send a CHALLENGE during the message exchange if the sender (i.e. *add to from* if *has add to* flag set, otherwise *from*) belongs to a different domain.

If Host B decides To issue a CHALLENGE, then it follows these steps:

1. Before continuing to download the remaining data on Connection 1, Host B MUST initiate a separate new connection (Connection 2) back to Host A using the same incoming IP address of Connection 1.
2. Host B sends a CHALLENGE to Host A, supplying the message header hash of the message header received on Connection 1.
3. Host A MUST verify the authenticity of the challenge by checking the header hash matches the message currently being sent to Host B. – If not matched then Host A MUST TERMINATE the message exchange.
4. Host A transmits a CHALLENGE RESP on Connection 2 consisting of the message hash.
5. Host B downloads the CHALLENGE-RESP which MUST be kept to later verify the message once fully downloaded.
6. Host A and Host B close Connection 2 and continue the message exchange on Connection 1.

### Notes on Challenge Mode

The automatic challenge is an important component of fmsg's message integrity and sender verification guarantees. So why the optionality, why not always automatically challenge if hosts need to implement handling it anyway? The intention is to allow trading protocol guarantees for efficiency which may be desirable depending on the use case.

The NEVER challenge mode discussed above could be useful on trusted private networks supporting a high volume of messages.

A HAS_NOT_PARTICIPATED challenge mode could be a useful middle ground where the first message in a thread has the extra checking and controls of an automatic challenge. The automatic challenge can help mitigate spam by performing strong sender verification and requiring the sender to listen, calculate the message digests and respond accordingly. Subsequent messages in a thread providing a valid pid where one of the messages in the thread is *from* Host B's domain provides proof of prior participation in the thread, which combined with checking the IP address is authorised for the domain already, gives a level of sender verification. Additionally, the receiving fmsg host could already have a level of message integrity guarantees, for example if the byte stream being read is over TLS. The combination of these guarantees might be sufficient for a recipient host to opt-out of challenging.

Challenging is particularly IMPORTANT for messages with *add to* recipients for Host B's domain. In that case, Host B may not have the parent message referenced by *pid* and therefore cannot verify it is stored – bypassing a check that normally provides thread integrity and an anti-spam signal. Without a challenge, such messages are effectively indistinguishable from unsolicited messages arriving for the first time.

Notification-only deliveries remain challengeable in the same way: an add-to message is a complete message header and the Sending Host holds the full message, so it can answer a CHALLENGE with the full message hash.

Ultimately, whether to challenge or not is at the discretion of the receiving host.

### *3. Continue, Per-Recipient Response and Disposition*

1. Host B determines and sends exactly one "REJECT or ACCEPT RESPONSE" code for the message header, evaluating the following checks in order – the first check that determines a code is the code sent, and the remaining checks are not performed:
    1. Iff *add to* exists and message was verified to be stored in step 1.4.6.3.3 above:
        1. If Host B has already recorded this exact add-to batch, i.e. the message hash for this batch matches a stored batch per Verifying Message Stored, Host B MUST respond REJECT code 10 (duplicate) then close the connection completing the message exchange.
        2. If any of the *add to* recipients are for Host B, Host B MUST respond with "ACCEPT or REJECT CODE" 65 (skip data) and message exchange continues. *NOTE* Host B has verified it already has message referred to by *pid* which means this message is an exact duplicate except for (*add to from*, *add to* and *time*). As on the code 11 path, Host B MUST record these fields exactly as transmitted so the batch's message hash can be faithfully recomputed per Verifying Message Stored – a subsequent reply may reference this batch by its hash.
        3. Otherwise none of the *add to* recipients are for Host B; Host B MUST respond with ACCEPT code 11 (accept add to) then close the connection completing the message exchange. *NOTE* Host B MUST record the new fields – *add to from*, *add to* and *time* – exactly as transmitted, along with the fact code 11 was sent, so the batch's message hash can be faithfully computed per Verifying Message Stored; subsequent messages may reference either the original message or this batch via *pid*. This is also the path taken by a **notification-only** participant domain – one with no address in this message's *to* or *add to* at all, being informed only that recipients were added.
    2. If the CHALLENGE, CHALLENGE-RESP exchange was completed, the message hash received in the CHALLENGE-RESP SHOULD be used to check if the message is already stored for **all** recipients on Host B per Verifying Message Stored. If the message is found to be already stored for all recipients on Host B, Host B MUST respond REJECT code 10 (duplicate) then close the connection completing the message exchange. *NOTE* If only some recipients still had the message stored continuing the message exchange allows restoring the message to those without the message.
    3. Otherwise, Host B responds with "ACCEPT or REJECT CODE" 64 (continue) and the message exchange continues.
2. If Host B responded with "ACCEPT or REJECT CODE" 65 (skip data), Host B MUST NOT read any further data from Connection 1. Otherwise (code 64), Host B continues

downloading the exact number of remaining bytes i.e. the sum of message *size* plus any attachments *size*. For each message or attachment data part with the corresponding *zlib-deflate* flag bit set, Host B MUST decompress the part and the decompressed byte count MUST exactly match that part's *expanded size*. If decompression fails, produces more bytes than *expanded size*, produces fewer bytes than *expanded size*, or otherwise does not exactly match *expanded size*, the message is invalid and Host B MUST TERMINATE the message exchange.

3. If the CHALLENGE, CHALLENGE-RESP exchange was completed, the message hash received in the CHALLENGE-RESP MUST exactly match the computed message hash per Computing Message Hash; otherwise Host B MUST TERMINATE the message exchange.
4. Host B transmits an "ACCEPT or REJECT RESPONSE" code to Host A for each individual recipient belonging to Host B.
    1. Host B iterates through each address for its domain (example.edu) in the order they appear in *to* then in *add to* (if any). An address in both lists is iterated twice and receives one code for its *to* entry and one for its *add to* entry. Note for any REJECT code specific to a user, 105 (user undisclosed) MAY be used instead – so Host B does not have to disclose the reason message was not accepted for that address. For each recipient:
        1. Host B looks up implementation specific data for the recipient address such as quotas and whether the address is accepting new messages.
        2. If the message has already been received for this recipient, Host B MUST respond either REJECT code 103 (user duplicate) OR REJECT code 105 (user undisclosed).
        3. If the address is unknown to Host B, Host B MUST respond either REJECT code 100 (user unknown) OR REJECT code 105 (user undisclosed).
        4. Else if accepting the message would exceed user quotas such as size or count limits, Host B MUST respond either REJECT code 101 (user full) OR REJECT code 105 (user undisclosed).
        5. Else if the address is not accepting new messages, Host B MUST respond either REJECT code 102 (user not accepting) OR REJECT code 105 (user undisclosed).
        6. Otherwise, Host B MUST respond ACCEPT code 200 (accepted) for the recipient address
    2. Host A MUST record the "ACCEPT or REJECT RESPONSE" code received per recipient for Host B's domain.
5. Host A and Host B close Connection 1, completing the message exchange.

*NOTE* When recipients for Host B are added using the *add to* functionality to a message Host B previously accepted, those previous recipients for Host B (that still have the message) would

respond REJECT code 103 (user duplicate), or code 105 (user undisclosed). Implementations SHOULD keep the original accept response code 200 to more accuratly reflect that status of the delivery to that recipient. Implementations MAY choose to record the additional response code as well.

## *4. Sending a Message*

A Sending Host (Host A) delivers a message if and only if *from* or *add to from* belongs to Host A's domain. When the *has add to* flag bit is not set, the message is sent to each unique recipient domain exactly once, regardless of how many recipients share that domain. When the *has add to* flag bit is set, the message is sent exactly once to each unique participant domain – the domains of *from* and of every address in *to* and *add to*, omitting *from*'s domain when *from* is the *add to from* (the adder is the original sender, whose host is the Sending Host) – so that all participants of the message being added to learn of the added recipients, not only the domains hosting the new recipients. This section describes the steps Host A performs for each domain. If multiple domains exist, Host A performs these steps independently for each domain without regard to the others.

Host A MUST NOT transmit a message whose *pid* references a message Host A holds with the *terminal* flag bit set, and SHOULD refuse to create such a message for its clients, see Notes on Terminal Messages.

1. Host A resolves the authorised IP addresses via Domain Resolution for Host B.
    1. Host A initiates a connection (Connection 1) to the first authorised IP address for the Receiving Host (Host B).
    2. If the first IP address is unresponsive within an implementation-defined timeout, and multiple IP addresses were returned during domain resolution, Host A SHOULD attempt to connect to each address in the order provided, one at a time, until a responsive host is reached.
    3. If no responsive Receiving Host is found, Host A SHOULD retry delivery after an implementation-defined delay. Implementations SHOULD apply a back-off strategy (e.g. exponential back-off) to subsequent retry attempts.
    4. Host A SHOULD continue retrying delivery until a maximum delivery window has elapsed, after which the message MAY be considered undeliverable for the affected recipients. Implementations MAY provide a mechanism for clients or operators to manually trigger or influence retry behaviour.
2. Before transmitting, Host A MUST register the *message hash* computed per Computing Message Hash of the outgoing message, and the IP address being used for Host B, both keyed on the *message header* hash. So that incoming CHALLENGE requests on Connection 2 can be matched to this message per Handling a Challenge.
3. Host A transmits the message header to Host B on Connection 1, encoding all fields in the order defined in Message:

1. *version*, *flags*, [*pid*], *from*, *to*, [*add to from*], [*add to*], *time*, [*topic*], *type*, *size*, [*expanded size*] and *attachment headers*.

4. Host A waits for Host B's response. During this time Host B may open Connection 2 to issue a CHALLENGE which Host A MUST handle per Handling a Challenge.
5. Host A reads the first byte from Host B on Connection 1 which represents a "REJECT or ACCEPT RESPONSE" code.
    1. If code is 1 through 10, the message being sent has been rejected for all recipients belonging to Host B. Host A MUST record the response then close Connection 1 completing the message exchange.
    2. If code is 11 (accept add to), and there were additional recipients in the message header, Host B – which hosts none of the *add to* recipients – has recorded the add-to batch and verified it already holds the message being added to, so no further transmission is needed. Host A MUST record the response then close Connection 1 completing the message exchange. Otherwise, if there were no additional recipients in the message header, Host A MUST TERMINATE the message exchange.
    3. If code is 64 (continue) indicating Host B instructs transmission of the message to continue (because they have validated the message is acceptable).
    4. If code is 65 (skip data) Host B instructs to skip sending message data and any attachments data (because they already have the data).
    5. Otherwise code is unrecognised, Host A MUST TERMINATE the message exchange.
6. If and only if "REJECT or ACCEPT RESPONSE" code was 64 (continue), Host A MUST transmit message data and any attachment data which MUST be of the exact length specified in *size* plus any attachment sizes.
7. Host A reads the next number of bytes as there were recipients for Host B on the outgoing message. Each byte represents a "REJECT or ACCEPT RESPONSE" code for each recipient belonging to Host B in the order they were on the message.
8. Host A MUST record the response code for each recipient.
9. Host A removes the *message header hash* from its outgoing record.
10. Host A and Host B close Connection 1, completing the message exchange for this recipient domain.

## Handling a Challenge

A Sending Host MUST be listening for incoming connections on the same IP address it uses to send outgoing messages. While a message is being transmitted, the Receiving Host may open a connection back to the Sending Host to issue a CHALLENGE. The Sending Host, Host A, handles this as follows:

1. Host A downloads the first byte
    1. If the value is less than 128 and a supported fmsg version – this is an incoming message and should be processed per Connection and Header Exchange.
    2. If the value is greater than 128 and 256 minus the value is a supported fmsg version, this is a CHALLENGE we support, continue.
    3. Otherwise Host A MUST TERMINATE the connection (unsupported version).
2. Host A downloads the next 32 bytes – the *header hash* supplied by Host B.
3. Host A MUST verify the authenticity of the challenge by checking:
    1. The *header hash* exactly matches a *message header hash* of a message Host A is currently transmitting.
    2. The IP address of the incoming connection challenging Host A is associated with the matched *header hash*. – If no currently outgoing *message header hash* AND associated IP address matches the supplied *header hash*, Host A MUST TERMINATE the connection. The challenge does not correspond to any message Host A is sending and may be spurious or malicious.
4. Host A responds with the *message hash* computed per Computing Message Hash included in CHALLENGE RESPONSE.
5. Host A and Host B close Connection 2. The message exchange continues on Connection 1.

*NOTE* A Sending Host MUST maintain a record of outgoing messages keyed by message header hash, including the IP address of each Receiving Host the message is being transmitted to. This record is used to match incoming challenges to the correct outgoing message and verify the challenger's IP address. The record MUST be created before transmission begins, as required by 4. Sending a Message. When a message exchange to a particular Receiving Host completes or is aborted, that host's IP address SHOULD be removed from the record. Once no IP addresses remain for a given message header hash, the entry SHOULD be removed.

## Computing Message Hash

The hash MUST be computed over the full message bytes, comprising the message header fields exactly as transmitted, followed by the message data and any attachments data. When the corresponding zlib-deflate flag is set for message data or each attachment's data, the data MUST be decompressed prior to inclusion in the hash computation. The decompressed byte count for each zlib-deflate part MUST exactly match the corresponding *expanded size* field before the computed hash can be considered valid; otherwise the message is invalid and the message exchange MUST be TERMINATED.

## Verifying Message Stored

A host MUST retain each stored message in full and exactly as transmitted – including the complete *to* and *add to* recipient lists, not only recipients belonging to its own domain. Both the hash

computation above and the participant checks in Protocol Steps step 1.4.6 depend on the message being reconstructible exactly as received.

A host verifies that a message is stored given a SHA-256 digest if:

- The provided digest exactly matches the SHA-256 digest computed per Computing Message Hash of a message that was either:
    - previously accepted by the host, i.e. for which the host responded with "REJECT or ACCEPT CODE" 200 (accept) to at least one recipient, OR "REJECT or ACCEPT CODE" 11 (accept add to); or
    - sent by the host, i.e. a message whose sender (*from*, or *add to from* when the *has add to* flag bit is set) belongs to the host's domain and which the host has transmitted or holds for sending – including each add-to message (batch) the host sent, since a reply may reference the batch by its hash.
- The corresponding message currently exists on the host and can be retrieved.

*NOTE I* An add-to message's data never crosses the wire when the receiving host already holds the message being added to (codes 11 and 65), and the sending host likewise already holds it. Computing an add-to message's hash therefore requires RECONSTRUCTING the message: combining the add-to message header exactly as transmitted (which has the add to fields) with the message and attachment data of the message referred to by *pid*. Every host holding a batch – whether it accepted it with code 11, accepted it with code 65 and per-recipient codes, or sent it – MUST be able to perform this reconstruction so replies referencing the batch by its hash can be verified. *NOTE II* Multiple messages with *add to* may arrive for the same *pid* over time, each carrying a batch of additional recipients. Only the specific batch that was accepted can be used for comparison. A batch's identity is its message hash, which covers *time*: two batches carrying the same *add to* addresses but different *time* values are distinct batches.

## Domain Resolution

Hosts MUST obtain and verify authorised IP addresses by resolving the subdomain `fmsg` of the domain name in an fmsg address and evaluating the resulting A and AAAA records (including those obtained via CNAME aliasing). For example if `@alice@example.com` is sending a message to `@bob@example.edu`, Alice's authorised fmsg host IP addresses are obtained by resolving `fmsg.example.com`, and Bob's from `fmsg.example.edu`.

Sending and receiving hosts SHOULD perform DNSSEC validation for fmsg lookups when supported. If DNSSEC validation fails, the connection MUST be terminated.

*NB* Before opening the second connection to send CHALLENGE, the Receiving Host MUST have independently resolved the senders authorised IP set from the `fmsg` subdomain and verified the originating IP address of the incoming connection is in that set. This would have been done in Protocol Steps 1.4.2. If verification fails the connection MUST be terminated without challenging. This ensures the fmsg host sending a message is listed by the senders domain and prevents

orchestrating a denial-of-service style attack by falsifying an address to trigger many fmsg hosts challenging an unsuspecting host.

### Notes on Domain Resolution

Various alternatives were considered before arriving at using the `fmsg` subdomain method. For instance an MX record combined with a WKS record on the domain would align with original intent of RFC 974 allowing message exchange services to be located for a domain along with WKS specifying the protocol. However the intent of MX records has been superseded and is now assumed to be SMTP and WKS is obsolete. Using a TXT record as SPF does was considered too, but that leads to a growing problem of proliferation of TXT records. A SRV record could have worked but the extra domain lookup after resolving a SRV record, to lookup the actual host, adds latency. So the `fmsg` subdomain method was chosen.

### Practical Concerns

Verifying the sender's IP address requires the Receiving Host to observe the true originating IP address of the connection. This implies that fmsg hosts must be directly routable, or that any intervening infrastructure preserves and conveys the originating IP address. Care must therefore be taken when fmsg hosts operate behind network address translators (NAT), layer-4 load balancers, or proxying infrastructure. This hard requirement is by design – tying sender’s to their domain which aides domain reputation.

## Security Concerns

This section identifies threats relevant to fmsg deployments and recommends safeguards for implementors and operators. Many of these overlap – a single malicious actor may exploit multiple vectors simultaneously – so a layered approach is strongly encouraged.

### Connection Flooding

An attacker can open many TCP connections to a host without sending valid message data, exhausting file descriptors, memory or connection-table capacity.

**Safeguards:**

- Hosts SHOULD enforce a maximum number of concurrent connections, in total and per source IP address.
- Hosts SHOULD apply short timeouts for idle or slow connections. If no valid version byte is received within a brief period the connection SHOULD be closed.
- Operating system level controls such as SYN cookies and connection rate limiting SHOULD be enabled.

## Oversized or Slow Data Attacks

A sender may begin transmitting a message header declaring a small size then either send data indefinitely, or send data extremely slowly ("slowloris" style), tying up resources.

**Safeguards:**

- Hosts MUST enforce the declared *size* plus *attachment sizes* and MUST NOT read beyond the expected total. MAX_SIZE provides an upper bound on transmitted data that should be checked before downloading any data.
- Hosts MUST enforce MAX_EXPANDED_SIZE before downloading any data by summing each uncompressed part's *size* and each compressed part's *expanded size*. For zlib-deflate parts, hosts MUST bound decompression output and terminate the message exchange if the decompressed byte count does not exactly match *expanded size*. Implementations SHOULD NOT buffer unbounded decompressed data solely to validate size or compute hashes.
- Hosts SHOULD enforce minimum data-rate thresholds. A connection where the transfer rate drops below a configurable floor for a sustained period SHOULD be terminated.

## Challenge Reflection and Amplification

A malicious host could forge the *from* address in a message to contain a victim's domain, causing the Receiving Host to open Connection 2 back to the victim as part of the automatic challenge – effectively using the Receiving Host as a reflector.

**Safeguards:**

- The specification already requires the Receiving Host to verify the originating IP address of Connection 1 against the DNS records for the sender's domain **before** issuing a challenge (see Domain Resolution). This is the primary defence and MUST NOT be skipped.
- Hosts SHOULD rate limit outgoing challenge connections per destination IP address to limit amplification even if DNS verification is somehow bypassed.

## DNS Spoofing and Cache Poisoning

If an attacker can poison DNS responses for `fmsg.<domain>`, they can redirect messages to a host they control or cause a legitimate host to accept messages from an unauthorised IP.

**Safeguards:**

- Hosts SHOULD perform DNSSEC validation for all `fmsg` lookups. If DNSSEC validation fails the connection MUST be terminated as specified in Domain Resolution.
- Hosts SHOULD use trusted resolvers and minimise DNS cache TTLs for `fmsg` records to reduce the window of exposure to poisoned entries.

## Message Replay

An attacker who has captured a valid message off the wire could attempt to re-deliver it to the same or different hosts.

**Safeguards:**

- The automatic challenge is the primary defence against replay: a replaying attacker cannot produce a valid CHALLENGE RESPONSE without possessing the original message in full and being resident at the authorised sender IP.
- Even without a challenge, hosts SHOULD maintain a record of recently accepted messages and reject duplicates (REJECT code 10 for all recipients, or per-recipient code 103).
- The time validity window (MAX_MESSAGE_AGE + MAX_TIME_SKEW) limits how long a captured message remains deliverable.

## Sender Enumeration

An attacker could craft messages addressed to many candidate recipient names to discover which addresses exist on a host, using REJECT code 100 (user unknown) vs 200 (accept) as an oracle.

**Safeguards:**

- Hosts MAY respond with REJECT code 105 (user undisclosed) for all per-user rejections, as the specification already permits. This prevents the attacker from distinguishing between unknown, full, and not-accepting addresses.
- Hosts SHOULD rate limit incoming messages from any single source IP or domain.

## Resource Exhaustion via Storage

An attacker could send a high volume of valid messages or very large messages to fill storage on the receiving host, preventing legitimate messages from being accepted.

**Safeguards:**

- Operators SHOULD configure MAX_SIZE and MAX_EXPANDED_SIZE to values appropriate for their deployment. MAX_EXPANDED_SIZE SHOULD normally be set to MAX_SIZE.
- Implementations SHOULD support per-user quotas on message count and total storage including over periods, e.g. daily. For compressed data, quotas SHOULD account for decompressed size. When quotas are exceeded the host responds REJECT code 101 (user full).
- Implementations SHOULD support global storage thresholds. When critically low the host responds REJECT code 5 (insufficient resources) to all incoming messages.

## Monitoring and Logging

Hosts SHOULD record the following for each message exchange to support detection and investigation of abuse:

- Timestamp
- Originating IP address
- When a message exchange was TERMINATED along with the reason why
- Domain from: *from* or *add to from* if exists
- REJECT or ACCEPT response codes sent
- Whether a CHALLENGE was issued and whether it succeeded

Operators SHOULD review these logs for anomalous patterns such as high rejection rates from a single domain or IP, repeated invalid messages, or spikes in connection volume. Automated alerting on such patterns is recommended.