fmsg

A message exchange protocol

Mark Mennell
@markmnl@fmsg.io
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 such as 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. Overall fmsg provides an efficient and secure messaging system with ownership and control at the domain level.

Table of Contents

Introduction

Electronic mail (email) has co-existed with The Internet since inception 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 for both. Feature wise comparison is made between email, IM applications and fmsg. Real world comparison is made sending and receiving various messages using fmsg and email. Appended are links to relevant code and document repositories, lists of IM applications and the fmsg specification.

Stronghold of Email

Email remains widely used today because it offers ownership and control at the domain level: messages travel via Domain Name Service (DNS) defined servers, server-to-server communication, allowing domain owners – especially corporations – to manage their own servers and meet

requirements around privacy, data sovereignty, cost, and legal obligations. Since nobody owns email itself, 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 longer, richer formatting options and its delayed, non-conversational nature. Another reason for email's persistence is 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. So to use many online services you must first have an email address.

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. As of writing spam, or "junk" email, constitutes nearly half of all emails sent[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. Since SMTP sends each email standalone, replies typically include the entire original email 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 practice of including original content in replies lacks a formalised 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. Attachments in email are sent as text – this alone increases the size of binary attachments by one third by base 64 encoding. 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, which if they are online will receive 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 two billion,

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

while WeChat (dominant in China) and Facebook Messenger each exceed one billion. Privacy concerned IM apps like Telegram and Signal are on the rise. Some IM applications are popular only in particularly regions of the world like KakaoTalk in South Korea and LINE in Thailand, Taiwan and Japan.

Group Messaging Platforms

Missing from the list of popular IM apps is a new 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”), they lack a standardised name so we refer to them 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 apps” with the exception of self-hosted platforms that are capable of hosting on one’s own infrastructure such as: Zulip, Rocket.Chat and Mattermost. An important distinction because when self-hosted, owners retain control over their infrastructure and data.

One notable exception of 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 other groups of users. Unlike those messaging 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 quite dissimilar to “IM apps” we refer to generally throughout this paper.

The fundamental difference between these Group Messaging Platforms (inc. self-hosted ones which includes Matrix) and fmsg is, *fmsg is just messages* – there are no forums/rooms/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 mail is standalone, the relationship of fmsg messages to previous 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 on the below Venn diagram. Note some features deliberately bleed across lines indicating their partial inclusion – for instance “prone to spam” is mostly in email and to a lesser extent in fmsg, “rich text formatting” is wholly in email and fmsg and only a bit in IM apps.

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

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 limited types.

Audio and video calls – Live audio and video transmission, only possible in IM apps usually using another protocol than the messaging one but initiated and bundled within the application.

Conversations friendly – Refers to the back and forth message replies between two or more participants. Email is not considered conversations friendly because each subsequent reply includes the full message history thus far – increasing resources needed and time to send and receive.

Message integrity – Are messages tamper proof in transit *and* at rest? Email itself has no integrity guarantees but can be configured to various degrees using additional mechanisms such as Transport Layer Security (TLS), DomainKeys Identified Mail (DKIM) and Domain-based Message Authentication, Reporting & Conformance (DMARC). Highlighting the operating complexity of email servers today because without these mechanisms email is vulnerable to tampering. fmsg has message integrity built-in such that even after the fact messages have been delivered they can verified they are unaltered. Proprietary IM apps cannot be independently verified but generally one would assume some message integrity mechanisms.

Multimedia messages – Pre-recorded audio and video files – a subset of arbitrary file attachments, is generally support 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 are owned and controlled at the domain level, whereas IM applications handle all users messages with their own servers. Worth noting: none of these are peer-to-peer between individual users – although theoretically email and fmsg can be if one uses their own domain. Practically operating one's own email server today is prohibitive due to the complexity and the extent of anti-spam measures email servers have to

enforce – giving rise to third party providers (Gmail, Outlook, Yahoo etc.) and resulting in the centralisation of email service providers. One of the motivations of fmsg is to enable owning and operating a service (at an organisation or even individual level) without relying on 3rd parties handling your messages.

Prone to spam – Whether the implementation would be susceptible to undesired messages typically sent in bulk for commercial, fraudulent or malicious purposes. Any service that allows unsolicited messages (a feature not a bug) is inherently prone. Email is particularly prone because of its wide adoption (a victim of it's own success), the lack of anti-spam measures in the initial design and the low cost of emailing.

Rich text formatting – Formatted text messages e.g. fonts, font size, emboldening, tables, lists etc. to provide control of presentation. Practically email supports a subset of Hyper Text Markup Language (HTML) for rich text formatting – both fmsg and email allow arbitrary Media Types but clients choose what to render inline. IM applications tend to support plain text with sprinkling of formatting varying on the application; though their emoji game is .

Sender Identify Verification – Verification messages are from who they say they are from. Email (SMTP) takes it at face value who the sender is which can be easily spoofed. So other mechanisms need to be deployed such as Sender Policy Framework (SPF) which checks the message came from the sender's domain. DKIM and DMARC previously mentioned for message integrity also provide domain level verification of the sender. fmsg includes all this domain level verification built-in such that a sender *proves* they are indeed the origin of a message during exchange; otherwise the message is not accepted.

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 do support unsolicited messages using a verified phone number to identify users, however, *IM applications are closed systems in that they only support messages amongst users of the same platform.*

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 direct acyclic graph (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.

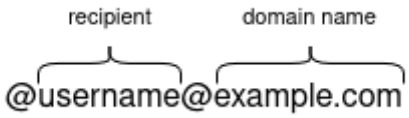


Figure 2: An fmsg address

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

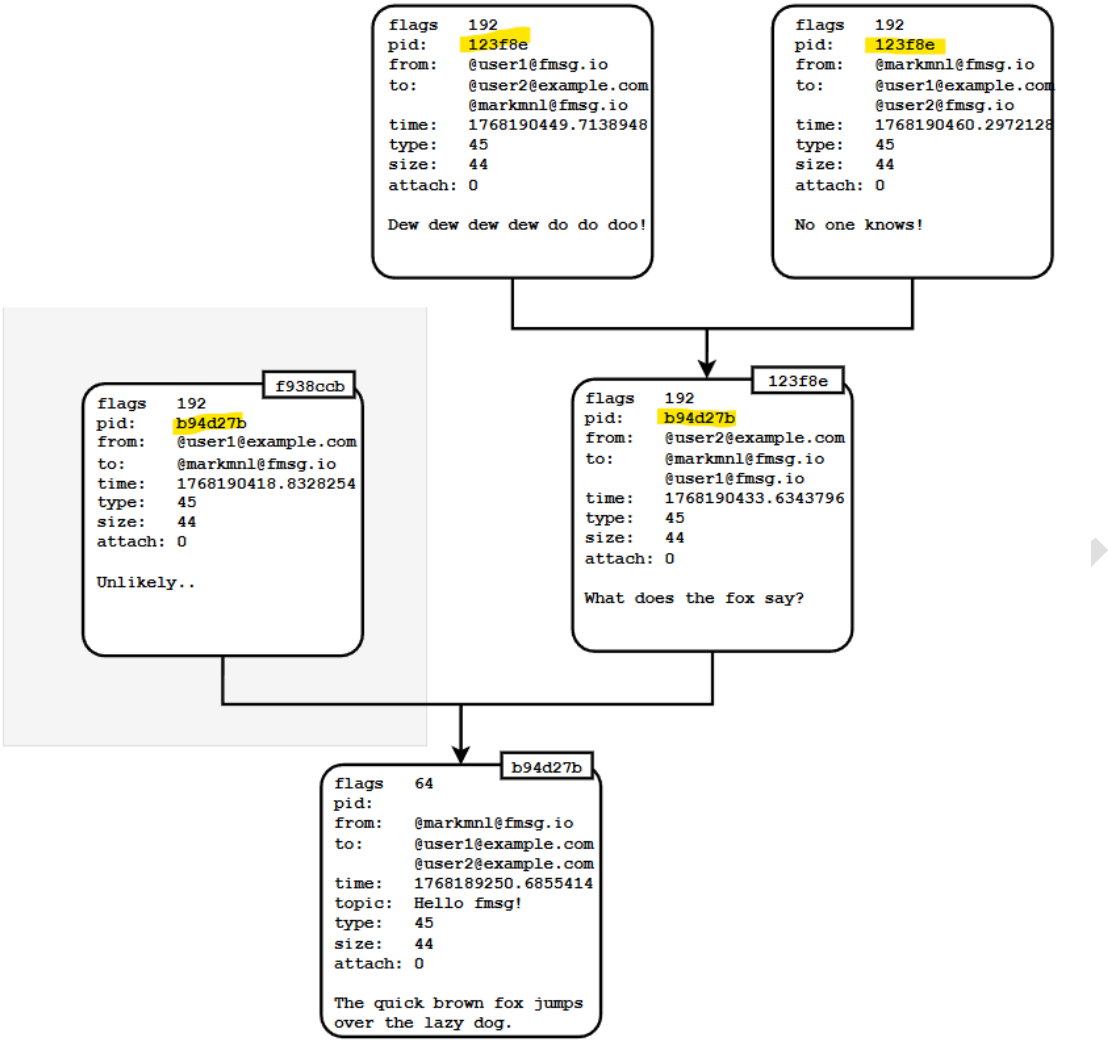


Figure 3: Message Thread Example

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 the protocol enforces during message-exchange 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

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. This separate challenge request provides some mitigation to "spamming" – because not only is the sender verified (at the domain level) but also requires a small cost in compute on the senders part – to deal with incoming connections and compute the hash of each message sent.

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.

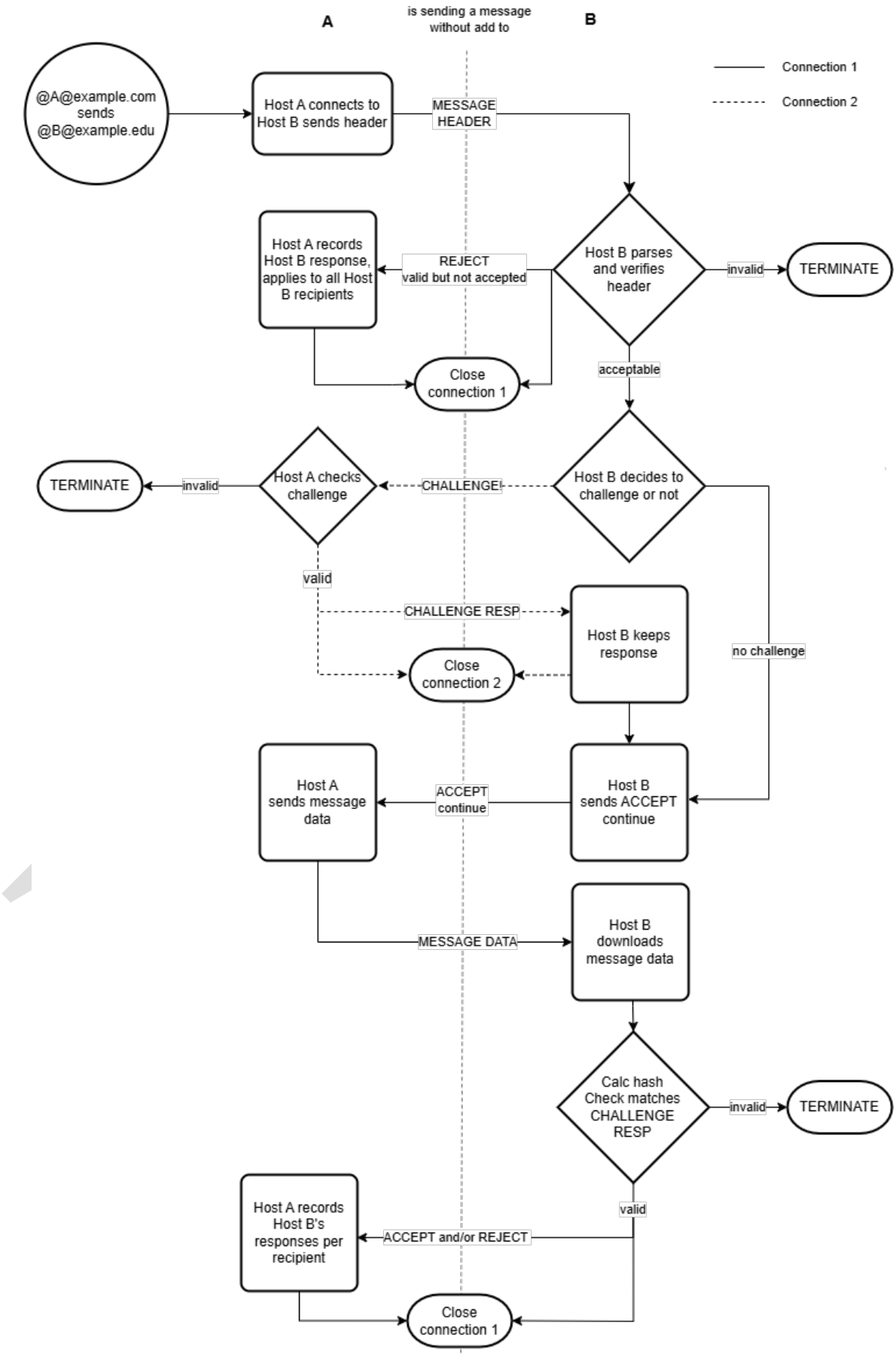
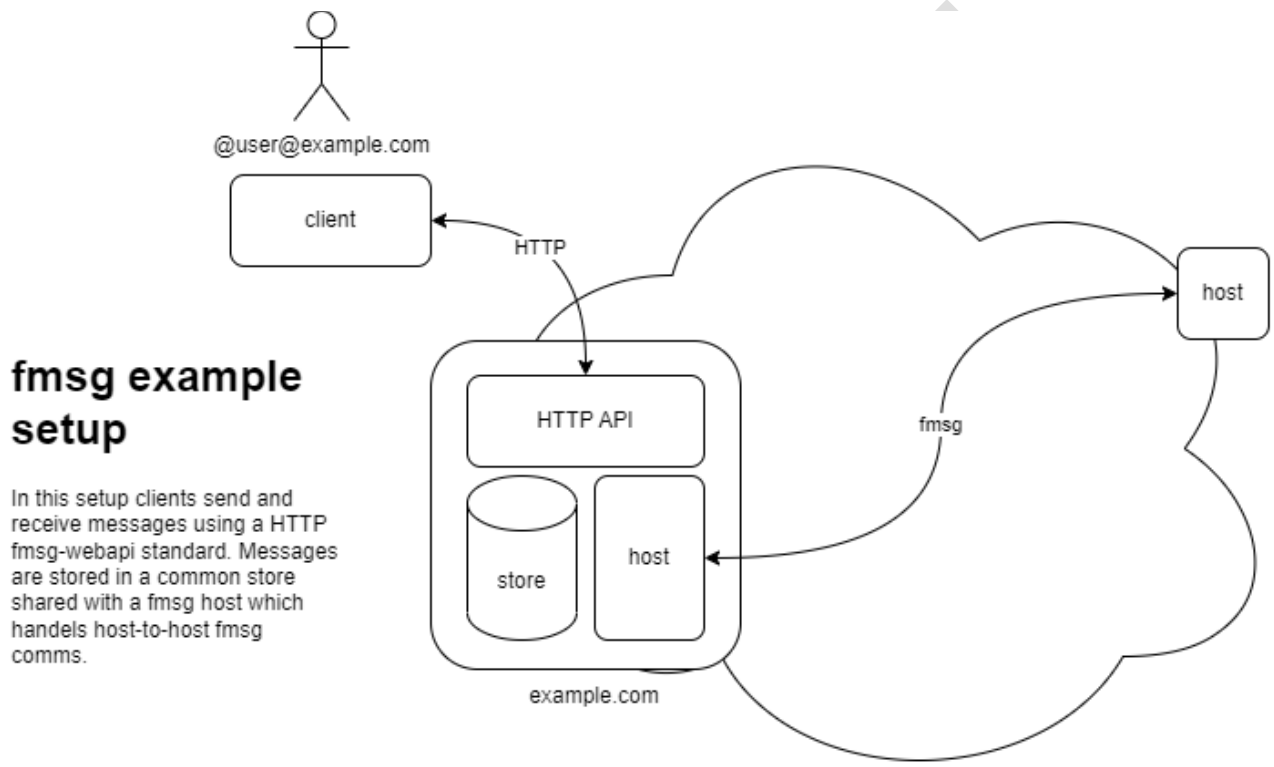


Figure 4: Protocol Flow Diagram

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.



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

We compare a fully functional setup of fmsg (using fmsg-docker[3]) vs. email (using mailcow[4]). Size of message transmitted and time taken to send are recorded over a variety of scenarios such as: simple one off messages, typical message exchanges, with/without attachments, forwarding, short conversations between two people, branching conversations amongst many people, etc.

3 fmsg-docker, https://github.com/markmnl/fmsg-docker, is a Dockerised stack composing a full fmsg setup including: fmsgd, fmsgid and fmsg-webapi.

4 mailcow, https://mailcow.email, is an open-source, all-in-one Docker-based email server suite developed and maintained by The Infrastructure Company GmbH, integrating services such as Postfix and Dovecot.

While care is taken to accurately record metrics fairly in a like-for-like manner these tests shouldn't be viewed as definitive "benchmarks" – especially with regards to timings which could vary between software stacks, versions, operating system etc. Also bear in mind there is a multitude of different email services and software of which only one has been picked here for comparison. Still, a realistic feel, especially for size of messages, in different scenarios can be gleaned from this comparison.

Method

All email and fmsg tests are conducted on the same infrastructure. Metrics are captured including TLS framing from fist SYN to final FIN:

- Total size of message transmitted between hosts in bytes (8 bit octets).
- Total time of transmission – which isn't just a function of bandwidth given the size because of real world implementations checking and controls – the built-in challenge-response of fmsg and various checks of a contemporary email setup.

Note, size and time of the client-host communication is not captured – which is dependent on the protocol used by the client to retrieve messages from the server. So we are really only testing email's core protocol; SMTP, vs. the fmsg protocol used for host-to-host transmission.

Command line tools *tcpdump* and *tshark* are used to capture and analyse tests. Actual test scripts run including commands to capture results are appended along with versions of software used so the tests can be replicated. While many more test scenarios could be concocted, the intent of the tests is to mimic typical real world messaging and get a feel how different message exchanges (with/without replies, multi-recipient, attachments etc.) affect the transmission size.

Table 1: List of tests

Test Name	Description
Sm1r	0
Sm1r1Sa	0
Sm4r	0
Sm4r1Sa	0
Sm1r1La	0
Sm4r1La	0
Sm1r_X	0
Sm1r1Sa	0
Sm4r	0
Sm4r1Sa	0
Sm1r1La	0
Sm4r1La	0

Results

TODO

Conclusion

TODO

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 Public Key Infrastructure, PKI, such as Transport Layer Security, TLS). Indeed the use of TLS 1.3+ is mandated by the only transport and binding defined so far[5].

5 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

Appendix I – Links

Documentation repository on GitHub: https://github.com/markmnl/fmsg, including "living" documents for:

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 2: Popular Instant Messaging Applications

Instant Messaging App	Approx. Monthly Active Users[6]	Company / Owner
WhatsApp	0	**0**
WeChat (Weixin)	0	**0**
Facebook Messenger	0	0
Telegram	0	**0**
Snapchat	0	**0**
QQ	0	0
Viber	0	**0**
LINE	0	0
Signal	0	**0**
KakaoTalk	0	**0**

Table 3: Popular Group Messaging Platforms

Platform	Typical Use Case	Self-hostable	Company / Owner
Slack	0		0
Microsoft Teams	0		0
Google Chat	Enterprise		
Zoom Team Chat			
Discord			
Zulip			

6 TODO

Appendix III – Real World Comparison Tests

Appendix III – Specification

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