Email Protocols Course

Flavio Copes

Updated August 6, 2026

Contents

Learn how email moves through SMTP, how MIME structures messages, how POP3 and IMAP retrieve mail, and where newer protocols fit.

What you'll learn: Trace an email from submission to mailbox, read SMTP replies and message headers, compare POP3 with IMAP, and choose a secure protocol for each stage.

How email moves

Follow a message through user agents, submission servers, relays, delivery agents, and mailboxes.

The parts of an email system

Name the agents that compose, submit, relay, deliver, store, and display one email message.

Email is not one protocol connecting two inboxes. It is a chain of specialized programs.

A **mail user agent** (MUA) is the app where you write and read mail. A **mail submission agent** (MSA) accepts outgoing mail from that app. **Mail transfer agents** (MTAs) relay it between domains. A **mail delivery agent** (MDA) places it in the recipient's mailbox.

The recipient then uses an access protocol such as IMAP or POP3, or a web application, to read the stored message. Learning the roles first prevents a common mistake: expecting one protocol to do every job.

Follow the ownership of one message:

> **Diagram: The parts of an email system.** A mail user agent submits a message to an MSA, two MTAs relay it, an MDA delivers it to the message store, and the recipient reads it with another MUA. Each arrow is a boundary where the message can be accepted, rejected, queued, or changed. A `250` from the MSA means the submission system accepted responsibility. It does not prove the receiving mailbox has the message.

The same product can implement several roles. A provider may run submission, relay, filtering, delivery, and storage behind one hostname. Keep the roles separate in your mental model even when the deployment combines them.

Draw the path for one message you send. Mark who owns the message after every successful handoff and where you would look for evidence if delivery stops.

The envelope and the message

Separate SMTP routing information from the From and To fields a person sees inside the message.

An email has an **envelope** and a **message**. They are related, but they are not the same data.

SMTP commands create the envelope. `MAIL FROM` supplies the return path and one or more `RCPT TO` commands name envelope recipients. The message begins later, after `DATA`, and contains visible

fields such as `From`, `To`, and `Subject`.

This separation makes forwarding, mailing lists, and blind copies possible. It also explains why debugging delivery starts with the envelope, while debugging what a user sees starts with the message headers.

Here is one complete distinction:

```
C: MAIL FROM:<bounces@news.example>
C: RCPT TO:<bob@example.net>
C: DATA
C: From: Alice <alice@example.org>
C: To: team@example.org
C: Subject: Private preview
C:
C: The release is ready.
```

The server routes toward `bob@example.net`. The reader sees `alice@example.org` and `team@example.org`. That is normal for a mailing list or blind copy. It is also why the visible `From` field alone does not identify the authenticated submitting account.

After final delivery, the receiving system can record the reverse path in `Return-Path`. Transport systems also prepend `Received` fields. Those fields become message trace data, but they do not change which envelope was used during the earlier SMTP transaction.

Take a saved `.eml` file and find `From`, `To`, `Return-Path`, and the oldest `Received` field. Write down which values came from the message and which describe transport.

Follow one email across the Internet

Trace a message from a mail app through submission, DNS routing, relays, delivery, and mailbox access.

You press Send. Your mail app submits the message to your provider, normally over an authenticated TLS connection.

The provider looks up the recipient domain's MX records in DNS. It opens an SMTP connection to a selected mail server, transfers the message, and queues a retry if that server reports a temporary failure.

The receiving system filters and delivers the message to a mailbox. The recipient reads that mailbox with IMAP, POP3, JMAP, a provider API, or the provider's web interface.

If the DNS step is unfamiliar, complete the DNS Course first. It explains MX records and mail-related TXT records in context.

The handoffs are store-and-forward:

```
mail app --submission--> sender MSA/MTA --relay--> receiver MTA --delivery--> mailbox
```

Suppose the receiving MTA replies `451 4.3.0 Temporary system problem`. The sending MTA keeps the message in its queue and retries later. If it replies `550 5.1.1 Mailbox unavailable`, that recipient normally fails permanently.

One message can have several recipients and several outcomes. The sending MTA might deliver to one domain, defer another, and reject a third. Delivery state belongs to each envelope recipient,

not only to the message as a whole.

The `Received` fields in the delivered message show successful SMTP hops. Read them from the bottom upward because each new server prepends its own field. A missing hop is not automatically malicious; internal systems may summarize or hide topology, and provider APIs may expose different trace data.

Draw three recipients on two domains. Give one recipient a `250`, one a `451`, and one a `550`. Record what the sender must do next for each address.

Protocols along the path

Place SMTP, MIME, POP3, IMAP, LMTP, Sieve, and JMAP at the correct points in an email journey.

SMTP submits and transfers mail. **RFC 5322** describes the basic message format. **MIME** adds content types, international text encodings, multipart bodies, and attachments.

POP3 downloads messages from a maildrop. **IMAP** synchronizes server-side mailboxes. **LMTP** is an SMTP-like final-delivery protocol, and **Sieve** is a server-side filtering language.

JMAP exposes synchronized mail data through HTTP and JSON. These technologies overlap at the edges, but each was designed for a different responsibility.

Put them on one path:

```
compose RFC 5322 + MIME message
  -> submit with SMTP
  -> relay with SMTP
  -> deliver with SMTP or LMTP
  -> filter with Sieve
  -> store in a mailbox
  -> access with POP3, IMAP, or JMAP
```

RFC 5322 and MIME are formats, not delivery sessions. IMAP and JMAP manipulate stored mail, but neither replaces the Internet SMTP relay between unrelated domains. Sieve runs near delivery; it does not need a desktop client online.

This distinction matters during failure analysis. A malformed MIME boundary can make an attachment unreadable even though SMTP delivery succeeded. An IMAP synchronization bug can hide a delivered message without changing its SMTP trace.

Classify these failures by layer: `550` during `RCPT TO`, an attachment that decodes incorrectly, and a message visible in webmail but missing on a phone.

Check your understanding: the email system

Check mail agents, the SMTP envelope, visible headers, DNS routing, storage, access protocols, and the complete delivery path.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The SMTP conversation

Read greetings, envelopes, message data, replies, and a complete SMTP transaction.

Start an SMTP session

Read the server greeting and use EHLO to identify a client and discover supported SMTP extensions.

An SMTP server speaks first with a `220` greeting. The client then introduces itself with `EHLO` and a domain name.

A successful `EHLO` reply begins with `250`. Additional `250-` lines advertise extensions such as `SIZE`, `STARTTLS`, `AUTH`, and `8BITMIME`.

```
S: 220 mail.example.net ready
C: EHLO client.example.org
S: 250-mail.example.net
S: 250-SIZE 52428800
S: 250 STARTTLS
```

Use the advertised capabilities instead of assuming every server supports the same extensions.

The greeting and `EHLO` establish session state. A multiline reply uses a hyphen after the code until the final line:

```
S: 250-mail.example.net
S: 250-PIPELINING
S: 250-SIZE 52428800
S: 250 STARTTLS
```

The last line uses a space, so the reply is complete. A client that stops reading after the first line can miss a size limit or try authentication before the server advertises it.

If the server rejects `EHLO`, an SMTP client can try `HELO` for the base protocol. It then loses extension negotiation. Submission software should not silently continue if the message requires an extension such as `SMTPUTF8`.

After `STARTTLS`, send `EHLO` again. Capabilities advertised before encryption are discarded, and the protected session may advertise a different set, including authentication mechanisms.

Annotate the transcript above. Mark the server identity, every extension, the final line, and the command that must be repeated after a TLS upgrade.

Build the SMTP envelope

Use MAIL FROM and RCPT TO to identify the reverse path and every envelope recipient.

A mail transaction begins with `MAIL FROM`. Its address is the reverse path used for delivery-status messages. A bounce uses an empty reverse path, written `<>`, to avoid bounce loops.

Each `RCPT TO` adds one envelope recipient. The server can accept one recipient and reject another before receiving the message body.

```
C: MAIL FROM:<alice@example.org>
S: 250 OK
C: RCPT TO:<bob@example.net>
S: 250 Accepted
```

Do not infer the envelope from the visible `From` and `To` fields. Inspect the SMTP transaction or delivery logs when routing is the question.

Recipient decisions are independent:

```
C: MAIL FROM:<alice@example.org>
S: 250 2.1.0 Sender accepted
C: RCPT TO:<bob@example.net>
S: 250 2.1.5 Recipient accepted
C: RCPT TO:<missing@example.net>
S: 550 5.1.1 Mailbox does not exist
```

The client can continue with `DATA` for Bob. It must not include the rejected address in the accepted recipient set. If no recipient is accepted, there is no useful message transaction to continue.

`MAIL FROM` can also carry advertised extension parameters such as `SIZE` or `SMTPUTF8`. `RCPT TO` can carry recipient-specific extension data. Send only parameters negotiated through `EHLO`.

Acceptance at `RCPT TO` is not final delivery. The receiving server can still reject the content after `DATA`, or accept it and later encounter a delivery failure that requires a status notification.

Extend the transcript with `DATA`, a short message, and a final `250`. Then list the exact recipients for which the server accepted responsibility.

Send message data

Enter DATA mode, terminate a message correctly, and understand why lines beginning with a dot are escaped.

After at least one recipient is accepted, the client sends `DATA`. A `354` reply means the server is ready for the message headers and body.

The client ends the data with `<CRLF>.<CRLF>`: a line containing only one dot. If a real message line starts with a dot, the client adds another dot. The receiving SMTP server removes it. This is called **dot-stuffing**.

The final `250` is important. It says the receiving server accepted responsibility for that message. A dropped connection before that reply leaves the sender uncertain and may cause a retry.

The transformation is visible on the wire:

```
Message line: .status is ready
Wire line:    ..status is ready
Terminator:   .
```

The server removes one leading dot from every dot-stuffed line. The terminator is not part of the RFC 5322 message. SMTP libraries handle this; application code should not add dots to the stored message itself.

After `354`, a normal server expects message content, not another SMTP command. If it detects a size or policy failure while receiving data, it reports the transaction result after the terminator. The client must read that final reply before it reuses the connection or removes its queued copy.

An uncertain disconnect is an at-least-once problem. The server may have committed the message just before the connection failed. A retry can produce a duplicate, so receivers and applications should not treat `Message-ID` as a perfect deduplication guarantee, but it can help identify repeats.

Write a five-line message containing a line that begins with a dot. Show the stored form, the wire form, and the final SMTP terminator separately.

Read SMTP reply codes

Use the first digit to distinguish progress, success, temporary failure, and permanent failure.

SMTP replies have three digits. `2xx` means success, `3xx` asks the client to continue, `4xx` is a temporary failure, and `5xx` is a permanent failure for the current request.

A sender normally queues and retries after `4xx`. Repeating the same message forever after `5xx` is not useful; the address, content, or policy must change.

Enhanced status codes add detail such as `5.1.1` for a bad destination mailbox. Read the text too, but automate decisions from the codes because server wording varies.

The command stage matters as much as the number:

```
S: 451 4.7.1 Try again later
S: 550 5.1.1 Mailbox unavailable
S: 552 5.3.4 Message too large
```

A `451` during `RCPT TO` defers that recipient. A `552` after `DATA` rejects the submitted content. Record the command, basic reply, enhanced status, and remote hostname together.

Do not classify every `4xx` as "the server is down." Greylisting, rate limits, mailbox storage, policy checks, and temporary DNS trouble can all defer mail. Back off instead of retrying in a tight loop.

Likewise, a `5xx` is permanent for the current attempt and parameters. It does not always mean the address can never receive mail. A too-large message can be changed; an unauthenticated relay attempt can be retried through the correct submission service.

For each reply above, decide whether to queue, change the message, change the route, or notify the sender. State what evidence led to the decision.

Check your understanding: SMTP

Check greetings, EHLO extensions, envelopes, multiple recipients, DATA, dot-stuffing, final acceptance, and reply-code families.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Submission and delivery

Separate submission from relay, negotiate extensions and TLS, authenticate, queue, and retry mail.

Submission is not relay

Distinguish a user submitting outgoing mail from servers relaying mail between domains.

Mail submission and server-to-server relay both use SMTP commands, but they have different trust and policy boundaries.

Port **587** is reserved for message submission. Clients normally authenticate there. Port **25** is the standard path for SMTP relay between mail servers, where the receiving server decides whether it is responsible for the destination domain.

Port **465** is the TLS-first submission service: the TLS handshake begins immediately. Do not describe 465 as server-to-server SMTP or confuse it with STARTTLS on an already-open connection.

Submission starts inside an account boundary. The MSA can require authentication, enforce which sender addresses the account may use, add missing fields, and apply rate limits. Relay on port 25 starts from another mail system and uses domain responsibility and anti-abuse policy instead of an end-user password.

```
587: TCP -> EHLO -> STARTTLS -> TLS -> EHLO -> AUTH -> MAIL
465: TCP -> TLS -> EHLO -> AUTH -> MAIL
25:  TCP -> EHLO -> MAIL -> RCPT -> DATA
```

The port does not create trust by itself. Certificate validation protects the server identity, authentication identifies the submitting account, and authorization decides which envelope senders and recipients that account may use.

Do not configure a user agent to submit through port 25 just because a connection succeeds. Networks often block it, and receiving MTAs should not behave like open submission services.

Capture the settings screen of a mail client without secrets. Identify the submission hostname, port, TLS mode, authentication identity, and whether the configuration matches one of the sequences above.

Negotiate ESMTP extensions

Read EHLO capabilities and understand how SMTP grows without silently changing its base protocol.

Extended SMTP, or ESMTP, uses the `EHLO` reply to advertise optional capabilities.

`SIZE` declares a message-size limit. `8BITMIME` supports an extended body format. `SMTPUTF8` permits internationalized addresses and headers when both sides agree. `PIPELINING` can reduce command round trips.

An extension is usable only when the server advertises it. A client must have a fallback or report that the message cannot be sent as constructed.

Capabilities change what the client may put on the wire:

```
S: 250-SIZE 10485760
S: 250-8BITMIME
S: 250 SMTPUTF8
C: MAIL FROM:<josé@example.org> SMTPUTF8 SIZE=2381
```

Without `SMTPUTF8`, the non-ASCII mailbox cannot be sent in that transaction. Without `8BITMIME`, a client must encode message bodies into a compatible representation or choose another route. `SIZE` lets the client avoid sending content the server has already said is too large.

`PIPELINING` changes timing, not command semantics. The client may send allowed commands without waiting for each reply, but it must still match replies in order and stop correctly after a failure. It must not pipeline commands the extension forbids.

Extensions are hop-by-hop. Support on the submission server does not prove the next relay supports the same feature. The sending system must choose routes and transformations that preserve the message or fail visibly.

Remove `SMTPUTF8` from the capability list above. Explain which part of the transaction becomes invalid and why changing only the visible `From` header would not fix the envelope address.

Protect and authenticate submission

Separate transport encryption, certificate validation, and user authentication in an SMTP connection.

TLS encrypts a connection and lets the client validate the server certificate. Authentication proves which account is submitting mail. They solve different problems.

On port 587, a client can issue `STARTTLS`, complete a TLS handshake, and send `EHLO` again because the protected connection has a fresh capability state. It should authenticate only after encryption is active.

On port 465, TLS begins immediately. Modern clients should not send passwords over a cleartext connection, and they should not ignore certificate errors just to make setup appear to work.

A protected submission exchange looks like this:

```
C: EHLO laptop.example
S: 250-STARTTLS
C: STARTTLS
S: 220 Ready to start TLS
... TLS handshake ...
C: EHLO laptop.example
S: 250-AUTH PLAIN OAUTHBEARER
```

The second `EHLO` is not optional cleanup. TLS resets the SMTP capability knowledge, and the server may advertise authentication only inside the protected channel.

An `AUTH` mechanism defines how credentials or tokens are exchanged. It does not encrypt the connection. Even a base64-looking credential is readable without TLS. Prefer short-lived tokens when the provider supports them, and never place secrets in transcripts or URLs.

Certificate validation checks that the certificate chains to a trusted authority, is valid for the submission hostname, and is within its validity period. A successful encryption handshake with the wrong host is not a safe connection.

Draw the state before TLS, during the handshake, and after the second `EHLO`. Mark the earliest point where authentication is allowed.

Queues, retries, and delivery status

Explain store-and-forward delivery, temporary deferrals, permanent failures, and delivery-status notifications.

SMTP is a store-and-forward protocol. A sending MTA can accept a message locally, place it in a queue, and try the next server later.

A `4xx` reply or unreachable destination normally keeps the message queued with increasing delays. A `5xx` reply normally ends delivery for that recipient.

If delivery ultimately fails after the original submission succeeded, the system sends a delivery-status message to the envelope return path. That notification itself uses an empty return path so another failure does not create an endless loop.

A queue record needs more than message bytes. It tracks accepted recipients, completed recipients, deferred recipients, the next attempt, and the last remote reply.

```
bob@example.net      delivered  250 2.0.0
team@example.net     deferred   451 4.4.1, retry at 14:30
missing@example.net  failed     550 5.1.1
```

Retries should spread out with backoff. A rapid loop makes a temporary incident worse and can trigger rate limits. The sender eventually reaches its queue lifetime, stops retrying, and creates a delivery-status notification for recipients still undelivered.

Do not generate a bounce to the visible `From` field. Delivery status follows the envelope reverse path. If that path is empty, as it is for another bounce, the failure is logged or discarded instead of generating a new message.

"Queued" is not "delivered." A submission API returning success may mean only that the provider accepted the job. Expose later delivery events separately in application status.

Create a queue table for three recipients using one success, one temporary failure, and one permanent failure. Write the next action for each row.

Check your understanding: submission and delivery

Check ports 25, 587, and 465, ESMTP discovery, TLS, authentication, queues, temporary failures, retries, and delivery status.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Messages and MIME

Understand headers, bodies, content types, encodings, multipart messages, and attachments.

Read an Internet message

Separate the RFC 5322 header section from the body and recognize continuation lines.

The message transferred after `DATA` has a header section, one empty line, and a body.

Each header field has a name, a colon, and a value. A field can continue on following lines that begin with whitespace, although generating unnecessarily folded fields is discouraged in modern software.

```
From: Alice <alice@example.org>
To: Bob <bob@example.net>
Subject: Project update
Date: Thu, 30 Jul 2026 10:00:00 +0200
Message-ID: <1234@example.org>

The deployment is complete.
```

The first empty line is structural. Everything before it is fields; everything after it is body content. A line in the body that looks like `Subject: hello` does not become a header.

Header field names are case-insensitive, but field meaning and allowed repetition come from their definitions. Parsers should unfold continuation lines before interpreting a field value. They must also reject raw line breaks injected through untrusted form input, or an attacker can create extra fields.

Message format uses CRLF line endings on the wire. Libraries normally normalize them. When investigating a malformed message, inspect the raw source rather than the rendered mail view because the viewer may repair or hide invalid structure.

Transport fields can be prepended after message creation. That means the delivered message is not necessarily byte-for-byte identical to the submitted one, even when its body is unchanged.

Add a folded `Subject` field and a body line beginning with `From:` to the example. Mark the exact empty line where header parsing stops.

Important message fields

Use originator, destination, date, subject, identifier, and trace fields without treating them as the SMTP envelope.

`From` identifies the message author. `Sender` can identify the agent that transmitted it on an author's behalf. `To` and `Cc` are displayed destinations; `Bcc` recipients should not appear in the delivered message.

`Date` is the message creation date. `Message-ID` is a globally unique identifier generated by the originating system. `Received` fields are added by transport systems and form a trace you normally read from bottom to top.

None of these replaces the SMTP envelope. The envelope controls the current delivery transaction.

Consider a mailing-list message:

```
From: Alice <alice@example.org>
Sender: Announcements <list@example.net>
To: Product list <product@example.net>
Message-ID: <launch-20260730@example.org>
```

`From` names the author. `Sender` says another agent sent it. The envelope return path may point to a per-recipient bounce address that appears in neither field.

`Message-ID` helps conversations, logs, and duplicate investigation, but it is not an authorization token or guaranteed proof of authorship. Authentication systems such as DKIM evaluate other evidence.

Each receiving SMTP server prepends its own `Received` field. Start with the bottom field to see the earliest recorded hop, then move upward. Compare timestamps with their numeric time zones and allow for clock mistakes.

Treat displayed fields as untrusted input. A friendly display name can contain misleading text, and `Reply-To` can direct responses somewhere other than `From`.

Inspect a raw message and write a table with author, sender, reply destination, message identifier, envelope return path, and oldest recorded hop.

MIME types and transfer encodings

Describe message content with media types, character sets, and safe transfer encodings.

MIME adds fields such as `MIME-Version`, `Content-Type`, and `Content-Transfer-Encoding`.

`Content-Type: text/plain; charset=utf-8` tells a reader how to interpret the bytes. `quoted-printable` keeps mostly readable text efficient; `base64` safely represents arbitrary binary data using ASCII characters.

Base64 is an encoding, not encryption. Anyone who receives it can decode it. The media type describes the content; the transfer encoding describes how that content was made safe for transport.

Here is a small text part:

```
Content-Type: text/plain; charset=utf-8
Content-Transfer-Encoding: quoted-printable

The caf=C3=A9 opens at 08:00.
```

The charset explains how decoded bytes become characters. The transfer encoding explains how those bytes were represented for transport. Mixing those two jobs produces mojibake or failed decoding.

Quoted-printable works well for text that is mostly ASCII. Base64 adds overhead but handles arbitrary bytes predictably. The decoder must process transfer encoding before it interprets the media type and charset.

Do not trust a declared `Content-Type` as proof of file safety. Senders can label executable content as an image. A receiving application should inspect content, apply size limits, and keep active data away from executable contexts.

Long lines and non-ASCII headers use their own message-format rules. Do not base64-encode an entire RFC 5322 message unless the containing protocol explicitly calls for it.

Decode `caf=C3=A9` in two steps: quoted-printable to bytes, then UTF-8 to text. Explain what breaks if you assume ISO-8859-1.

Multipart messages and attachments

Use MIME boundaries to carry alternative bodies, related resources, and file attachments.

A multipart body contains several parts separated by a boundary declared in its `Content-Type` field. Each part has its own MIME headers and body.

`multipart/alternative` offers representations of the same content, commonly plain text followed by HTML. `multipart/mixed` groups independent parts, commonly a message body and attachments.

`Content-Disposition: attachment` suggests that a part should be presented as a downloadable file. A filename is presentation metadata, not proof that the bytes are safe. Applications must still validate and scan attachments.

A minimal multipart message looks like this:

```
Content-Type: multipart/mixed; boundary="mail-boundary-42"

--mail-boundary-42
```

```
Content-Type: text/plain; charset=utf-8

The report is attached.
--mail-boundary-42
Content-Type: application/pdf
Content-Transfer-Encoding: base64
Content-Disposition: attachment; filename="report.pdf"

JVBERi0xLjQK
--mail-boundary-42--
```

The closing delimiter has two extra hyphens. The boundary value must not occur inside a part. Real MIME structures can nest: `multipart/alternative` for text and HTML can sit inside `multipart/mixed` beside attachments.

Clients usually prefer the last representation they understand in `multipart/alternative`, but every representation should express the same message. Do not put tracking or essential content only in the HTML part.

Sanitize filenames before writing them. Remove path components, choose your own storage name, limit decoded size, and scan the decoded bytes. A filename such as `../../report.pdf` must never control a filesystem path.

Extend the example with a nested plain-text and HTML alternative. Draw the MIME tree before writing the boundaries.

Check your understanding: messages and MIME

Check headers and bodies, trace fields, envelope separation, content types, charsets, transfer encodings, multipart alternatives, and attachments.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

POP3

Download messages through POP3 states, commands, identifiers, deletion rules, and secure connections.

The POP3 model

Understand POP3 as a small protocol for accessing a maildrop, commonly by downloading messages to a client.

POP3 gives a client access to a server maildrop. Its classic workflow is simple: authenticate, list messages, retrieve them, optionally mark them for deletion, and quit.

The protocol has three states. **Authorization** identifies the user. **Transaction** works with messages. **Update** applies deletions when the client quits successfully.

POP3 can leave copies on the server, but it does not provide IMAP's rich shared mailbox model. It fits best when one client owns a downloaded archive.

The states control which commands are legal:

```
TCP/TLS connected -> AUTHORIZATION -> TRANSACTION -> UPDATE -> closed
```

The server starts with `+OK` or `-ERR`. Successful authentication enters transaction state. `STAT`, `LIST`, `UIDL`, `RETR`, and `DELE` operate there. `QUIT` commits deletion marks and closes the session.

POP3 message numbers are session positions, not durable identifiers. If another session changes the maildrop, "message 4" can refer to a different position next time. Use `UIDL` values to remember which messages were already downloaded.

A client should write the retrieved message durably before it sends `DELE`. Otherwise a local crash can lose the only copy after a successful update.

The server commonly locks the maildrop for a transaction. A second client may receive an authentication or lock failure until the first session finishes.

Write the valid state beside `USER`, `PASS`, `STAT`, `RETR`, `DELE`, and `QUIT`. Then explain what should happen if the TCP connection dies before update.

Authenticate and inspect a maildrop

Read POP3 greetings and use STAT, LIST, and UIDL after authentication.

A POP3 server greets with `+OK` or fails with `-ERR`. After a protected connection is established, a simple session may use `USER` and `PASS` to enter the transaction state.

`STAT` returns the message count and total size. `LIST` returns message numbers and sizes. `UIDL` returns server-assigned unique identifiers that clients can remember between sessions.

Message numbers are temporary positions in the current maildrop. Use unique IDs, not positions, to avoid downloading the same message repeatedly.

Watch the reply forms:

```
S: +OK POP3 server ready
C: USER alice
S: +OK User accepted
C: PASS ...
S: +OK Maildrop has 2 messages
C: STAT
S: +OK 2 4812
C: UIDL
S: +OK Unique-ID listing follows
S: 1 whqtswO00Q430
S: 2 QhdPYR:00WBw1
S: .
```

`STAT` is one line. `UIDL` without a message number is multiline and ends with a dot line. As with SMTP, a real response line beginning with a dot is dot-stuffed.

`USER` and `PASS` do not protect credentials. Run them only inside a TLS-protected connection whose certificate you validated. A server can also advertise stronger authentication mechanisms through POP3 extensions.

Store the UIDL only after the corresponding message is durable locally. Servers are expected to

keep a unique ID stable while that message remains in the maildrop, but a client still needs recovery logic for server bugs or a rebuilt mailbox.

Parse the transcript into message count, total octets, and UID map. Identify which reply tells you the multiline section is complete.

Retrieve and delete messages

Use RETR, DELE, RSET, and QUIT while respecting POP3 update semantics.

`RETR 3` downloads message 3. `DELE 3` marks it for deletion; it does not necessarily erase the message at that instant.

`RSET` clears deletion marks during the transaction. A successful `QUIT` moves the session into update and commits marked deletions. If the connection dies before update, the server should not apply them.

This transaction model protects against some interrupted downloads, but the client still needs durable bookkeeping before it deletes the server copy.

A safe sequence is explicit:

```
C: RETR 3
S: +OK 1842 octets
S: ...message lines...
S: .
C: DELE 3
S: +OK Message marked
C: QUIT
S: +OK Deletions committed
```

First parse and store the entire multiline response. Verify that local storage completed. Only then mark the server copy for deletion.

`DELE` changes session state, not necessarily disk state. `RSET` removes the marks before `QUIT`. If update fails, the server can leave some or all marked messages undeleted; the client must tolerate seeing them again.

`TOP` can retrieve headers and a limited number of body lines when supported, but it is optional. Do not use a partial preview as proof that the full message was archived.

Deletion policies such as “leave mail on server for 14 days” are client behavior layered over POP3. Several clients applying different policies can produce surprises because POP3 has no shared folder or flag model.

List the crash points before local save, after local save, after `DELE`, and during `QUIT`. State whether the next session can safely redownload the message at each point.

Secure POP3 and know its limits

Use TLS for mailbox access and decide when POP3 is too limited for synchronized devices.

Cleartext mailbox access exposes credentials and messages. Modern email access should use TLS with certificate validation. POP3 over implicit TLS commonly uses port 995.

POP3 is intentionally narrow. It has no standard hierarchy of synchronized folders, server-side flags

such as `\Seen`, or rich server searches comparable to IMAP.

Choose POP3 for a deliberate download/archive workflow. Choose IMAP or JMAP when several devices need one consistent server-side view.

With implicit TLS, the first application bytes arrive only after the TLS handshake:

```
TCP 995 -> TLS handshake -> +OK POP3 greeting -> authentication
```

Validate the hostname and certificate chain. “Use SSL” with validation disabled only hides an active interception or configuration error.

POP3's simplicity moves synchronization work to the client. It does not standardize folders, sent-mail upload, read flags, conflict resolution, or server-side search. UIDL helps prevent duplicate downloads, but it is not a complete multi-device state model.

Leaving copies on the server can preserve access from another device, yet each client still decides independently which messages are new and when to delete them. One client's update can surprise the others.

If a legacy server offers cleartext POP3 only, place replacement or a trusted tunnel on the roadmap. Do not send reusable credentials across an untrusted network.

Compare a phone and laptop using one POP3 account. Write down who owns read state, sent mail, deletion timing, and the archive. Then repeat for IMAP.

Check your understanding: POP3

Check POP3 states, greetings, STAT, LIST, UIDL, retrieval, deletion marks, QUIT behavior, TLS, and the download-oriented model.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

IMAP

Work with server-side mailboxes, flags, searches, stable identifiers, synchronization, and push updates.

The IMAP model

See IMAP as a protocol for manipulating server-side mailboxes rather than merely downloading messages.

IMAP keeps the authoritative mailbox on the server. Clients synchronize mailbox names, messages, flags, and changes.

A message can appear with flags such as `\Seen`, `\Answered`, `\Flagged`, `\Deleted`, and `\Draft`. Several clients can observe the same state.

That model is more complex than POP3, but it matches how people use phones, laptops, and webmail with one account.

An IMAP connection moves through states:

`not authenticated -> authenticated -> selected mailbox -> logout`

Authentication gives access to the account. `SELECT INBOX` enters selected state, where message commands such as `FETCH`, `STORE`, and `SEARCH` become valid. `CLOSE` or `UNSELECT` leaves that mailbox without ending the account session.

The server can send untagged updates at any time. A client cannot assume that only its own actions change message counts or flags. Another phone, a server-side filter, or a delivery process can update the selected mailbox.

Flags are shared state, not local decoration. Setting `\Seen` on one client can make the message appear read everywhere. Use UIDs and mailbox state to merge changes rather than uploading one device's entire view over another.

IMAP accesses mail; it does not submit it. A mail app still uses SMTP submission or a provider API to send outgoing messages, then usually stores a sent copy through IMAP or provider behavior.

Draw the state transitions for login, `LIST`, `SELECT`, `UID FETCH`, `CLOSE`, and `LOGOUT`. Mark one command that is invalid before a mailbox is selected.

Tagged commands and selected mailboxes

Read tagged client commands, untagged server data, completion responses, and mailbox selection.

Every IMAP client command begins with a tag chosen by the client, such as `A001`. The final server response repeats that tag so several outstanding commands can be matched.

Untagged responses begin with `*` and carry data or state changes. A continuation request begins with `+`.

After authentication, `LIST` discovers mailboxes and `SELECT INBOX` opens one for read-write access. `EXAMINE` opens a mailbox read-only.

Tags make concurrency readable:

```
C: A001 LIST "" "*"
S: * LIST (\HasNoChildren) "/" "INBOX"
S: A001 OK LIST completed
C: A002 SELECT INBOX
S: * 42 EXISTS
S: * OK [UIDVALIDITY 93842] UIDs valid
S: A002 OK [READ-WRITE] SELECT completed
```

The untagged lines are data. `A001 OK` and `A002 OK` complete their matching commands. `NO` means a valid command could not be performed; `BAD` means a protocol or syntax problem.

Mailbox names and hierarchy delimiters come from the server. Do not split every mailbox name on `/`; the `LIST` response states the delimiter, and a `NIL` delimiter means no hierarchy is available.

`SELECT` returns synchronization evidence such as message count, `UIDVALIDITY`, and often `UIDNEXT`. Cache those values with the mailbox. A successful selection of the same display name does not prove its UID generation stayed the same.

Add a second command tag before `A001` completes. Then match every final response to its command and separate untagged data from completion status.

Fetch, store, and search

Retrieve selected message data, change flags, and ask the server to search a mailbox.

`FETCH` requests specific data rather than forcing the whole message to download. A client can ask for headers, body sections, flags, sizes, or dates.

`STORE` changes flags. `SEARCH` finds messages matching criteria such as sender, date, subject, or flag state.

```
C: A003 UID SEARCH UNSEEN
S: * SEARCH 104 108
S: A003 OK Search completed
```

Server-side search reduces transfers, but clients must still handle unsolicited changes reported while the session is open.

Use `UID` commands for durable work:

```
C: A004 UID FETCH 104 (FLAGS BODY.PEEK[HEADER.FIELDS (FROM SUBJECT)])
S: * 7 FETCH (UID 104 FLAGS () BODY[HEADER.FIELDS (FROM SUBJECT)] {52}
S: From: Alice <alice@example.org>
S: Subject: Deployment
S:
S: )
S: A004 OK FETCH completed
```

The `7` is the current sequence number. `104` is the UID. `BODY.PEEK` retrieves data without setting `\Seen`; the equivalent non-PEEK body fetch can change shared flag state.

`STORE` should express the intended change. `+FLAGS` adds flags and `-FLAGS` removes them. Replacing the complete flag list can erase a change made by another client.

Search results are identifiers matching the query at that moment. A later expunge or delivery can change the mailbox before the client fetches them. Handle missing results and unsolicited updates instead of assuming a frozen snapshot.

Fetch only what the interface needs. Headers and small body sections make an inbox fast; download large attachments after the user asks for them.

Change the example to mark UID 104 as seen without replacing its other flags. Explain why using sequence number 7 in a later session is unsafe.

UIDs, synchronization, and IDLE

Use stable identifiers and mailbox state to synchronize safely while receiving near-real-time updates.

IMAP sequence numbers are positions and can change when messages are expunged. UIDs are stable within one mailbox and UIDVALIDITY generation.

A synchronizing client records UIDs and mailbox state, fetches changes, and rebuilds local state when UIDVALIDITY changes. It should not treat a sequence number as a durable message ID.

The `IDLE` extension lets a server report changes without constant polling. TLS-protected IMAP commonly uses port 993.

A local cache key should include the account, mailbox, `UIDVALIDITY`, and UID. A UID alone is not globally unique and can be reused after the mailbox generation changes.

```
C: A010 IDLE
S: + Idling
S: * 43 EXISTS
C: DONE
S: A010 OK IDLE terminated
```

`DONE` is not a tagged IMAP command. It ends the idle period associated with `A010`, whose tagged completion arrives afterward. Clients leave and re-enter IDLE periodically according to server and network behavior.

An `EXISTS` update says the selected mailbox count changed. It does not contain the new message. The client still fetches new UIDs and the data it needs.

When `UIDVALIDITY` changes, discard UID-based assumptions for that mailbox and rebuild. When a connection drops, reconnect, reselect, compare saved state, and ask the server for changes; do not assume every pushed update was received.

Design a cache record for one message. Then write the recovery steps after the network disappears while the client is idling.

Check your understanding: IMAP

Check mailboxes, flags, tagged and untagged responses, selection, fetching, searching, sequence numbers, UIDs, synchronization, and IDLE.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Choose the right email protocol

Compare POP3 and IMAP, then place LMTP, Sieve, JMAP, and provider APIs in the email system.

Choose POP3 or IMAP

Choose between a download-oriented maildrop and a synchronized server-side mailbox.

Use POP3 when one client should download a simple maildrop and local storage is the intended archive. Its small command set can be an advantage.

Use IMAP when several clients need shared folders, flags, searches, partial fetching, and synchronized state. Most modern personal mail setups fit this model better.

Neither protocol sends mail. A client commonly combines SMTP submission with POP3 or IMAP access.

Choose from the ownership model, not from a port number.

Question	POP3	IMAP
Authoritative copy	Commonly local after download	Server mailbox
Shared folders and flags	No standard model	Yes
Durable identifier	UIDL per maildrop	UID within UIDVALIDITY
Partial fetch and search	Limited	Built in

Question	POP3	IMAP
Multi-device conflicts	Client policy	Protocol synchronization

POP3 can be correct for an appliance that downloads one maildrop into a controlled archive. IMAP is usually correct for a person who expects phone, laptop, and webmail to share read state and folders.

Security is not the differentiator: both should run with TLS and certificate validation. Complexity is. IMAP exposes more server state because it solves a larger synchronization problem.

Migration needs care. A POP3 client may hold the only copy of older messages. Moving to IMAP does not automatically upload that local archive.

Choose a protocol for a legal archive collector, a personal mailbox on three devices, and a monitoring inbox processed by one daemon. State where the authoritative copy lives in each design.

LMTP and Sieve

Place local delivery and server-side filtering after SMTP transport without confusing them with mailbox access.

LMTP resembles SMTP but is designed for final delivery where a server can report a separate result for each recipient after receiving the content. It is often used between a mail server and a local mailbox system.

Sieve is a small server-side language for filtering messages. Rules can file, redirect, reject, or discard mail without depending on a desktop client staying online.

LMTP moves a message into its final delivery system. Sieve decides what to do with it. Neither is a reading protocol.

LMTP changes the result after message data. SMTP returns one transaction result, while LMTP returns one result for each accepted recipient:

```
C: LHLO mx.example
C: MAIL FROM:<alice@example.org>
C: RCPT TO:<bob@example.net>
C: RCPT TO:<full@example.net>
C: DATA
S: 354 Send message
C: ...message...
C: .
S: 250 2.1.5 bob delivered
S: 452 4.2.2 full mailbox temporarily over quota
```

That per-recipient result fits a final-delivery hop where mailbox outcomes are already known. LMTP is commonly used on a trusted internal connection or local socket, not as the public Internet relay protocol.

A Sieve rule works on message metadata and supported tests:

```
require ["fileinto"];

if header :contains "subject" "Invoice" {
```

```
  fileinto "Finance";
}
```

Filtering occurs server-side, so every client sees the result. Rules should have a safe default and avoid loops when redirecting mail.

Give the LMTP transcript one permanent recipient failure. Then write a Sieve rule that files alerts without discarding unmatched messages.

JMAP for mail

Understand why JMAP uses HTTP and JSON for efficient synchronization and API-friendly mail clients.

JMAP defines a general synchronization protocol over HTTP and JSON. RFC 8621 applies it to email objects, mailboxes, searches, changes, and submission.

It can batch several method calls into one request, use state tokens for incremental synchronization, and receive push updates instead of polling constantly.

JMAP is not SMTP with JSON syntax. It is an application protocol that can replace much of an IMAP-style client interface while existing mail servers still use SMTP for Internet transfer.

A JMAP client first discovers the session object. It learns the API URL, accounts, capabilities, upload URL, and download URL instead of guessing endpoints.

Method calls can be chained in one request:

```
{
  "using": ["urn:ietf:params:jmap:core", "urn:ietf:params:jmap:mail"],
  "methodCalls": [
    ["Email/query", { "accountId": "a1", "filter": { "isUnread": true } }, "q1"],
    ["Email/get", { "accountId": "a1", "#ids": { "resultOf": "q1", "name": "Email/query", "pa
  ]
}
```

The second call uses IDs produced by the first. Batching reduces round trips while keeping each method result explicit.

State strings support incremental synchronization. If the client presents stale state to a change method, it must follow the defined recovery path rather than overwriting newer server data.

JMAP runs over HTTPS, so normal HTTP authentication, TLS validation, status handling, and request-size limits still matter. The JSON body can contain method-level errors even when the HTTP request itself succeeded.

Identify the query call, result reference, and get call above. Add the state value your local cache would need for a later incremental sync.

Trace and choose the complete email path

Map every stage of a real provider setup and choose protocols according to trust, synchronization, and interoperability needs.

Draw one outgoing path: mail app → authenticated submission → sending MTA → DNS MX lookup → receiving MTA → delivery → mailbox → reading client.

Label the protocol at every arrow. Include the ports and TLS mode. Mark where the SMTP envelope ends and where the RFC 5322/MIME message begins.

Provider HTTP APIs and webhooks can be excellent application interfaces, but they are provider contracts rather than universal mail transport. Keep SMTP at the interoperability boundary unless both ends explicitly share another system.

Finally, capture one real message's headers and identify its `Message-ID`, MIME type, and `Received` trace without publishing addresses, tokens, or private content.

Build the trace as an evidence table:

Stage	Protocol or format	Success evidence	Common failure evidence
Submission	SMTP on 587 or 465	final `250` and queue ID	`535`, `552`, TLS error
Relay	SMTP on 25	remote final `250`	`4xx`, `5xx`, timeout
Delivery	SMTP or LMTP	mailbox delivery result	quota or policy reply
Storage	provider mailbox	UID or object ID	missing or rebuilt state
Access	IMAP, POP3, JMAP	tagged OK, `+OK`, method result	protocol or sync error

Do not use one layer's success as proof for the next. A webhook stating “accepted” may correspond to local queueing. A delivered event may still precede client synchronization. Define status names from the evidence you actually receive.

Redact raw traces before sharing them. Authentication tokens, internal hostnames, mailbox addresses, message content, and unique provider IDs can all be sensitive. Keep timestamps, reply classes, and a stable internal correlation ID where possible.

Trace one test message to an account you control. Record a submission ID, the oldest and newest `Received` fields, the envelope result if available, the MIME tree, and the mailbox identifier. Explain every gap instead of filling it with a guess.

Check your understanding: choosing email protocols

Check POP3 versus IMAP, LMTP, Sieve, JMAP, provider APIs, SMTP interoperability, and the complete secure email path.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/email-protocols/.