FTP Course

Flavio Copes

Updated August 6, 2026

Contents

Learn how FTP uses control and data connections, how commands and replies work, why active and passive modes differ, and how to transfer files safely.

What you'll learn: Read an FTP session, transfer files safely, diagnose active, passive, firewall, and TLS problems, and choose the right file-transfer protocol.

How FTP works

Understand the client, server, control connection, data connections, sessions, and reply codes.

What FTP is

Understand FTP as a stateful client-server protocol for navigating remote filesystems and transferring files.

FTP stands for **File Transfer Protocol**. It predates the Web and was designed for interactive file work between a client and server.

An FTP client can authenticate, change the remote working directory, list entries, download files, upload files, rename them, and remove them when permitted.

The protocol remains interesting because it exposes its operations as readable commands and numeric replies. It also carries historical design choices that do not fit modern firewalls easily.

FTP is stateful. The server remembers the authenticated user, remote working directory, transfer type, and next data endpoint across commands.

```
C: USER alice
S: 331 Password required
C: PASS ...
S: 230 Logged in
C: PWD
S: 257 "/uploads" is the current directory
```

Commands and replies stay on a control connection. Listings and file bytes use separate data connections. That split is the defining operational detail of FTP and the source of many firewall failures.

A successful command does not make classic FTP safe. Without TLS, credentials, filenames, commands, and file contents are visible and modifiable in transit.

Connect only to a server you own or are authorized to use. Follow one listing and one download in verbose client output. Mark the control replies and the separate data connection.

Control and data connections

Explain why an FTP session keeps commands separate from directory listings and file contents.

FTP uses a long-lived **control connection** for commands and replies. It normally reaches the server on TCP port 21.

Directory listings and file contents travel over separate **data connections**. A new data connection is normally created for each listing or transfer, then closed.

This separation lets commands remain responsive while data moves, but it means one apparent operation involves more than one TCP connection. Most FTP troubleshooting begins by asking which connection failed.

One listing uses both channels:

> **Diagram: FTP control and data connections.** The client requests passive mode on the control connection, opens a separate data connection for the listing, reads its bytes, then waits for the final completion reply on the control connection. The data connection has no FTP commands. Its bytes mean whatever the control command selected: a listing for `MLSD`, downloaded content for `RETR`, or uploaded content for `STOR`.

Closing the data socket is not final success. The client must still read `226` or another completion reply on the control connection. A server can consume all bytes and then report a filesystem error.

Draw the two TCP connections for this exchange. Label who initiates each connection, which reply is preliminary, and which reply proves completion.

FTP session state and roles

Follow the user and server protocol interpreters, transfer processes, login state, and working directories.

RFC 959 names a user protocol interpreter, server protocol interpreter, and data transfer processes. In practical terms, the client sends commands while both sides coordinate the data connection.

The server keeps session state: authenticated user, current directory, transfer type, and selected active or passive data endpoint.

A command can therefore depend on earlier commands. FTP is not a series of independent requests like a simple stateless HTTP download.

The dependency is visible in a short session:

```
C: TYPE I
S: 200 Type set to I
C: CWD releases
S: 250 Directory changed
C: EPSV
S: 229 Entering Extended Passive Mode (|||50022|)
C: RETR app.tar.gz
```

`RETR` now means "retrieve `releases/app.tar.gz` as image bytes through the selected passive endpoint." A reconnect loses that context. The client must authenticate and establish it again.

Some commands create a smaller state machine. `RNFR old.txt` selects a rename source; only a following `RNTO new.txt` completes the operation. A different command or failure can invalidate that sequence.

Concurrent automation should use separate FTP sessions. Sharing one control connection lets one task change the working directory, transfer type, or data endpoint underneath another.

Write down the session state after each line above. Then reconnect and list which commands must be repeated before the same retrieval is valid.

Read FTP reply codes

Use the three digits in an FTP reply to understand completion, connection, authentication, and filesystem results.

FTP replies have three digits followed by text. The first digit describes the broad result.

`1xx` means an operation has started and another reply will follow. `2xx` means completion. `3xx` needs more information. `4xx` is a temporary failure. `5xx` is a permanent failure for the request.

The second digit groups the subject: syntax, information, connections, authentication, or filesystem. For example, `150` opens a data transfer and `226` confirms that the transfer finished.

Read replies in command context:

```
C: STOR release.zip.part
S: 150 File status okay; opening data connection
...bytes move...
S: 452 Insufficient storage space
```

The preliminary `150` did not promise success. The final `452` says the upload failed temporarily after transfer started. Keep the local source and do not rename the remote temporary file into place.

Common evidence includes `220` for a service greeting, `331` for another authentication value, `425` when the data connection cannot open, `530` for login or account policy, and `550` for an unavailable file action.

Multiline replies use the same code with a hyphen until the final line uses a space. Read through the terminator before sending a decision based on an incomplete capability response.

For every failed operation, record the command, numeric reply, text, transfer mode, and whether a data connection opened. Classify the example above without reducing it to "upload failed."

Check your understanding: how FTP works

Check FTP responsibilities, session state, control and data connections, port 21, transfer lifecycle, 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

Commands and files

Log in, navigate directories, list entries, download, upload, rename, and delete files.

Connect and log in

Read the greeting and use USER and PASS without mistaking protocol examples for a safe cleartext login.

The server normally greets a new control connection with `220`. The client sends `USER`; a `331` reply asks for a password, and `230` confirms login.

```
S: 220 FTP server ready
C: USER alice
```

```
S: 331 Password required
C: PASS ...
S: 230 User logged in
```

Classic FTP sends these commands over cleartext. Use this transcript to understand the protocol, not as a recommendation to expose credentials on an unprotected network.

The greeting can also defer or reject the connection. `120` asks the client to wait for a later greeting; `421` means the service is unavailable and the server will close the control connection.

Authentication is a sequence:

```
C: USER alice
S: 331 User name okay, need password
C: PASS ...
S: 530 Login incorrect
```

After `530`, the session is not authenticated. Do not continue with file commands or retry passwords rapidly. Check the exact account, server policy, and lockout behavior.

With explicit FTPS, send `AUTH TLS` and complete certificate-validated TLS before credentials. With SFTP, none of this transcript applies because SFTP authenticates through SSH and is a different protocol.

Use a test account with narrow permissions. Capture the greeting and login result without recording the password. Mark the reply that changes the session into authenticated state.

Navigate remote directories

Use PWD, CWD, and CDUP while keeping local and remote working directories distinct.

`PWD` asks the server for the current remote directory. `CWD path` changes that directory. `CDUP` moves to its parent.

Interactive clients often provide local commands too, such as `lcd`. Those are client features and do not travel over FTP.

When a relative path fails, inspect both the local and remote current directories before changing permissions or assuming the file disappeared.

The remote server interprets FTP paths:

```
C: PWD
S: 257 "/incoming" is the current directory
C: CWD reports/2026
S: 250 Directory changed
C: PWD
S: 257 "/incoming/reports/2026" is the current directory
```

The visible path can be a virtual filesystem path, not a host operating-system path. A server may confine the account to a root and hide everything above it.

Quote paths through the client correctly, but remember that shell quoting and FTP pathname syntax are different layers. A GUI may send the command for you; verbose output reveals the actual remote path.

A `550` after `CWD` can mean missing path, denied traversal, or a server policy. Confirm `PWD`, test each

path segment, and inspect server logs before changing broad permissions.

Create two nested directories on a test server, navigate with relative and absolute paths, then prove the local working directory did not change.

List files and metadata

Compare human-oriented LIST output with NLST names and standardized MLSD facts.

`LIST` asks for a directory listing, but its text format was never standardized well enough for reliable machine parsing. `NLST` returns names with less detail.

`MLSD`, defined later, returns standardized facts such as type, size, and modification time. A capable automated client should prefer MLSD when the server advertises it with `FEAT`.

Every listing still uses a data connection. A successful login followed by a hanging listing is therefore usually a data-channel problem, not an authentication problem.

Machine-readable facts look like this:

```
type=file;size=4812;modify=20260730101522; report.csv
type=dir;modify=20260729120000; archive
```

`MLSD` returns a directory over the data connection. `MLST path` can describe one object in a control reply. Servers advertise supported commands and facts through `FEAT`.

Treat modification time as metadata, not identity. Clocks, upload tools, and server policy can preserve or rewrite it. Size plus time still cannot prove that two files have equal content.

If only `LIST` is available, use a mature client parser and expect server-specific formats. Do not split on spaces and assume the final token is always a filename.

Capture `FEAT`, then request one listing. Record whether the client chose `MLSD`, `NLST`, or `LIST`, and explain how it parsed a filename containing spaces.

Download, upload, and change files

Recognize the core commands for retrieving, storing, appending, renaming, and deleting remote files.

`RETR path` downloads a file. `STOR path` uploads or replaces one. `APPE path` appends data when supported.

`DELE` removes a file. Renaming is a two-command sequence: `RNFR old-name` selects the source, then `RNTO new-name` supplies the destination.

The server replies on the control connection while file bytes use the data connection. Wait for the final completion reply before treating the operation as successful.

An upload should use a temporary remote name:

```
C: STOR release.zip.part
S: 150 Opening data connection
...bytes...
S: 226 Transfer complete
C: RNFR release.zip.part
S: 350 Ready for destination name
C: RNTO release.zip
```

```
S: 250 Rename successful
```

The `226` proves the transfer command completed. The later `250` proves the rename completed. Keep those as two separate facts.

`STOR` can replace an existing object depending on server policy. Do not assume a failed upload preserved the old file unless you tested the server behavior. `APPE` is especially dangerous when a retry repeats bytes.

Before destructive commands such as `DELE`, capture the exact remote path and account. Avoid building commands from untrusted filenames without validation.

Run this workflow against disposable files. Interrupt one upload before `226` and confirm that the public filename never exposes the partial content.

Check your understanding: commands and files

Check login, remote navigation, listings, MLSD, downloads, uploads, appends, renames, deletions, and final completion replies.

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

Take this interactive quiz online

Active and passive FTP

See who opens each data connection and why NAT, IPv6, and firewalls change the result.

Active FTP

See how PORT tells the server where to open the next data connection back to the client.

In classic active mode, the client listens for a data connection and sends a `PORT` command containing its address and port. The server opens the connection back to that endpoint.

This direction was natural when clients were directly reachable. It is awkward when a laptop sits behind NAT or a firewall that blocks unexpected incoming connections.

Active mode changes only the data channel. The client still opened the control connection to port 21.

The endpoint is encoded in `PORT`:

```
C: PORT 192,0,2,44,195,80
S: 200 PORT command successful
C: RETR report.csv
```

The port is `195 * 256 + 80`, which is 50000. The server then tries to connect to `192.0.2.44:50000` for the transfer.

That address must be reachable from the server. A client behind NAT often advertises a private address or cannot accept the inbound connection. The control login still works, which makes the later `425` look confusing.

Servers should reject `PORT` endpoints that target unrelated hosts or unsafe ports. Otherwise the server can be abused as a relay in an FTP bounce attack.

Trace one active transfer on a test network. Identify the advertised client endpoint, the server's outbound connection attempt, and the final reply after the data socket closes.

Passive FTP

See how PASV lets the client open both control and data connections to the server.

With `PASV`, the server listens on a temporary data port and returns the address and port in a `227` reply. The client opens the data connection there.

Because the client initiates both connections, passive mode usually works better through client-side NAT and firewalls.

The server firewall must permit its configured passive port range, and the advertised address must be reachable by the client. A bad advertised private address is a classic failure.

A classic passive reply carries six numbers:

```
C: PASV
S: 227 Entering Passive Mode (203,0,113,10,195,81)
```

The client connects to `203.0.113.10` on port `195 * 256 + 81`, or 50001. It opens that data connection before or around the following transfer command, depending on the client flow.

NAT must map the advertised public endpoint to the server. The FTP service should use a narrow configured passive range so firewall rules remain predictable.

Some clients ignore an unsafe address in `227` and reuse the control connection's peer address. Do not depend on that repair. Configure the server to advertise the right public address, or prefer `EPSV`, which returns only a port.

Decode the reply above by hand. Then compare the address with the control connection peer and decide whether a remote Internet client can reach it.

EPRT and EPSV

Use the extended active and passive commands designed for IPv6 and simpler address handling.

`EPRT` generalizes the active-mode endpoint so it can represent different network protocols, including IPv6.

`EPSV` returns only a port because the client normally connects to the same server address used by the control connection. A reply can look like `229 Entering Extended Passive Mode (|||6446|)`.

EPSV avoids parsing an address supplied inside the reply and is the natural passive command for IPv6.

An extended active command names the address family explicitly:

```
C: EPRT |2|2001:db8::44|50002|
S: 200 EPRT command successful
```

The delimiter is chosen by the client. `2` means IPv6, followed by the network address and TCP port.

Extended passive mode is simpler:

```
C: EPSV
S: 229 Entering Extended Passive Mode (|||50003|)
```

The client connects to port 50003 on the same server address as the control connection. There is no embedded IPv4 address for NAT to rewrite.

A server can reject an unsupported network protocol with `522` and indicate supported families. The client should fall back deliberately, not parse a failed reply as an endpoint.

Write the connection tuple produced by each example and mark who opens the data connection.

NAT, firewalls, and failed transfers

Diagnose an FTP session that logs in successfully but cannot list or transfer files.

If login works, the control connection works. If `LIST`, `RETR`, or `STOR` hangs, focus on the separate data connection.

In active mode, check whether the server can reach the client endpoint. In passive mode, check the server's passive port range, public address advertisement, NAT mapping, and firewall rules.

Capture the FTP replies and identify whether the failure happens before `150`, during transfer, or before final `226`. "FTP is down" is not a useful diagnosis until you identify the channel.

Use a layered trace:

```
1. DNS and TCP reach the control service
2. 220 greeting arrives
3. authentication reaches 230
4. PASV or EPSV returns an endpoint
5. the data TCP connection opens
6. 150 starts the operation
7. bytes and EOF cross the data channel
8. 226 completes the command
```

If step 4 advertises `10.0.0.8` to an Internet client, fix server NAT configuration. If step 5 times out, inspect passive-range mappings and firewalls. If bytes stop after `150`, inspect network resets, storage, and timeouts. If `426` replaces `226`, the transfer was aborted.

FTPS adds another layer: firewalls cannot inspect encrypted control commands to open ports dynamically. Configure and permit an explicit passive range instead of relying on an FTP-aware helper.

Reproduce one failure with verbose client logs and server logs. Stop at the first missing step in the list and change only that layer.

Check your understanding: active and passive FTP

Check PORT, PASV, EPRT, EPSV, connection direction, NAT, firewalls, passive port ranges, and data-channel diagnosis.

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

Take this interactive quiz online

Transfer behavior

Choose transfer types, inspect metadata, resume safely, and avoid exposing partial uploads.

ASCII and binary transfer types

Use TYPE I for byte-for-byte transfers and understand the historical newline conversion performed by TYPE A.

`TYPE A` is ASCII mode. It can translate text line endings between the local system and the network representation.

`TYPE I` is image mode, commonly called binary mode. It transfers bytes without text newline conversion and is the safe default for images, archives, executables, PDFs, and most modern files.

Using ASCII mode on a binary file can corrupt it. Modern clients commonly select binary automatically, but verify the setting in automation and old interactive clients.

The client sets transfer representation before the data command:

```
C: TYPE I
S: 200 Type set to I
C: RETR database.sqlite
```

Image mode does not mean the file is an image. It means an uninterpreted sequence of bytes. Use it for nearly every modern transfer, including UTF-8 text, unless a legacy system explicitly requires network-text conversion.

ASCII mode can map local line endings to FTP's network representation and back. A retry across different systems can therefore produce content with different bytes even when the visible text looks similar.

Size and restart offsets can depend on the selected representation. Set `TYPE I` before comparing byte sizes or resuming a binary transfer.

Transfer one small text file in both modes between systems with different newline conventions. Compare cryptographic hashes and explain the difference.

The lifecycle of one transfer

Follow a listing or file operation from data-endpoint selection through preliminary and completion replies.

First the client selects active or passive mode. Then it sends a transfer command such as `RETR`.

A preliminary reply such as `150` says the server is opening the data connection. Bytes move on that connection. Closing it marks the end of data, and the control connection receives a final reply such as `226`.

A complete data stream without a successful final reply is not enough. The server may have detected a write, policy, or filesystem error after receiving bytes.

The control transcript should bracket the data operation:

```
C: EPSV
S: 229 Entering Extended Passive Mode (|||50010|)
C: RETR archive.tar.gz
S: 150 Opening BINARY mode data connection
...data socket reaches EOF...
S: 226 Transfer complete
```

The client can fail at several independent points: endpoint negotiation, TCP connection, preliminary

authorization, byte transfer, or final server completion.

For downloads, write to a local temporary file. Rename it only after the expected byte count or hash and the final successful reply. For uploads, apply the same pattern on the remote side when the server supports rename.

A timeout after data EOF but before 226 is uncertain. The server may have completed the operation. Inspect remote state before blindly retrying an append or destructive action.

Interrupt the example at each stage and predict the visible local and remote files. Verify the prediction against a disposable server.

Inspect size, time, and resume support

Use SIZE, MDTM, and REST carefully to inspect and resume transfers when the server advertises them.

`SIZE` can report a file size. `MDTM` can report its last modification time in UTC. `REST STREAM` allows a later transfer to restart at a byte marker when supported.

These are extensions, so discover them with `FEAT`. Modification time is useful metadata, but it is not a content checksum.

Before resuming, confirm the remote object and local partial file still refer to the same content. Resuming the wrong version creates a quietly corrupted result.

A resumed download can look like this:

```
C: TYPE I
S: 200 Type set to I
C: SIZE archive.tar.gz
S: 213 104857600
C: REST 52428800
S: 350 Restarting at 52428800
C: RETR archive.tar.gz
```

The client appends bytes starting at the marker to a 52,428,800-byte local partial file. A mismatch between the marker and local length corrupts the output.

`MDTM` timestamps use UTC in the standardized form. They help detect a changed object but remain weaker than a content digest. FTP has no universally deployed standard command that guarantees a cryptographic hash.

Resume support can differ by command and transfer type. Discover features, use `TYPE I`, and test the exact server. If identity cannot be established, restart from zero.

Create a partial test file, resume it, and compare the final cryptographic hash with a known-good copy. Then change the remote file and show how size-only validation can miss the wrong content.

Upload without exposing partial files

Use temporary names, completion checks, renames, and independent hashes to make uploads safer.

Uploading directly to a public filename can expose a partial file to another process. A safer pattern is to upload to a temporary name and rename it only after a successful transfer.

Wait for the final FTP completion reply. When integrity matters, compare a cryptographic hash

through a trusted side channel or supported server extension; file size alone cannot prove equal content.

FTP rename semantics depend on the server filesystem, so test the workflow you depend on. The protocol does not make a multi-file deployment atomic.

Treat publication as a small state machine:

```
local source
  -> STOR release.zip.part
  -> final 226
  -> verify size or trusted hash
  -> RNFR release.zip.part
  -> RNTO release.zip
  -> final 250
```

Use a unique temporary name so two uploaders do not overwrite the same partial object. Clean abandoned temporary files with a bounded retention policy, not by deleting every name ending in `.part`.

A hash protects integrity only when the expected digest arrives through a trusted path. A hash downloaded beside the file over the same compromised cleartext FTP session does not detect an attacker who can replace both.

The final rename may be atomic on one server filesystem, but FTP does not promise that across directories, mounts, or storage backends. A group of files also needs a release pointer, manifest, or server-side deployment mechanism if readers must see one consistent version.

Run the sequence with a deliberate mid-upload disconnect and a deliberate hash mismatch. Confirm that neither failure replaces the public file.

Check your understanding: transfer behavior

Check ASCII and binary modes, transfer replies, SIZE, MDTM, REST, resuming, temporary names, final replies, and integrity checks.

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

Take this interactive quiz online

FTP security

Understand plaintext risks, anonymous access, FTPS, certificate checks, and why SFTP is different.

Plaintext FTP risks

Understand what an observer can learn or change when FTP credentials, commands, filenames, and file data are not encrypted.

Classic FTP does not encrypt the control or data connection. A network observer can read usernames, passwords, commands, filenames, listings, and file contents.

An active attacker may also alter transferred data or replies. Restricting filesystem permissions helps the server, but it does not protect traffic in transit.

Do not expose cleartext authenticated FTP to the public Internet. Use a protected protocol and validate the server identity.

Both channels leak:

```
control: USER alice
control: PASS reusable-password
control: CWD payroll
control: RETR july.csv
data:    employee and salary records
```

Protecting only the password would not hide filenames or file contents. Protecting only the control connection would still leave listings and transfers exposed.

Cleartext also lacks integrity. An on-path attacker can alter a downloaded installer, replace an upload, or change a reply. A successful `226` proves only what the FTP server observed on that connection.

Use explicit FTPS with private data connections when FTP interoperability is required. Prefer SFTP, HTTPS, or a purpose-built storage API when you control both ends. Restrict credentials and filesystem access even after encryption because a compromised account still acts with its granted permissions.

Write a threat table for the transcript above. List what a passive observer learns, what an active attacker can change, and which risk remains after TLS.

Anonymous FTP and bounce attacks

Treat anonymous publishing as a narrow policy choice and understand why unrestricted PORT commands are dangerous.

Anonymous FTP can publish public downloads without individual accounts. It should expose only intended directories and should not combine public uploads with public execution or downloads carelessly.

The classic FTP bounce attack abuses `PORT` to make a server open a data connection to another host and port. Servers should restrict data endpoints and avoid connecting to unrelated addresses or privileged ports.

Old protocols accumulate security lessons in later RFCs. Reading RFC 959 alone is not enough to deploy FTP safely.

Anonymous access is still an account and policy boundary. Give it a fixed root, read-only access to intended public files, rate limits, logging, and no access to secrets or executable server paths.

If anonymous uploads are unavoidable, separate upload and download directories. Use server-generated names, quotas, scanning, and a review step before publication. Otherwise the service can become public file hosting or a malware exchange.

A bounce attempt asks the FTP server to connect somewhere chosen by the client:

```
C: PORT 198,51,100,20,0,25
```

That encodes port 25 on another host. A safe server rejects endpoints that do not match the control client's address and blocks privileged or otherwise unsafe destinations.

Passive mode reduces this specific direction of abuse because the client connects to the server. It

does not make anonymous uploads safe by itself.

Review a test server's anonymous root and active-mode policy. Prove that an anonymous user cannot upload executable content or make the server connect to an unrelated address.

Protect FTP with TLS

Upgrade the control connection with AUTH TLS and protect data connections with PBSZ and PROT.

FTPS is FTP secured with TLS. In explicit FTPS, the client connects to the normal FTP service and sends `AUTH TLS`. After a successful reply, the TLS handshake protects the control channel.

The client uses `PBSZ 0` for TLS and `PROT P` to require private, encrypted data connections. Protecting only the control channel would still expose file contents and listings.

The client must validate the server certificate. Encryption without identity validation can connect securely to the wrong server.

The control upgrade is a state transition:

```
C: AUTH TLS
S: 234 AUTH TLS successful
...TLS handshake and certificate validation...
C: PBSZ 0
S: 200 PBSZ=0
C: PROT P
S: 200 Data protection set to Private
```

Send credentials only after the handshake. `PBSZ 0` is required for the TLS protection mechanism. `PROT P` requests confidentiality and integrity for later data connections.

Every passive data connection gets its own TLS protection. That increases handshake and firewall complexity. Reusing the protected control connection does not encrypt a separate data socket automatically.

Do not silently fall back when `AUTH TLS` or `PROT P` fails. A client configured to require FTPS should stop instead of completing the same transfer in cleartext.

Trace one protected listing with verbose output. Mark the control handshake, `PROT P`, passive endpoint, data-channel handshake, and final `226`.

FTPS is not SFTP

Distinguish FTP over TLS from the SSH File Transfer Protocol and choose configuration vocabulary precisely.

FTPS preserves FTP commands, replies, and separate data connections, then protects them with TLS.

SFTP is the SSH File Transfer Protocol. It runs as an SSH subsystem, commonly over one SSH connection on port 22. It does not use FTP commands, active mode, or passive mode.

A hosting dashboard that asks for SFTP credentials is not asking you to enable FTPS. Match the client mode to the server protocol; similar names do not make them interchangeable.

Compare the connection models:

```
FTPS: control TLS connection + separate TLS data connections
SFTP: SSH connection + SFTP subsystem messages in that connection
```

FTPS authenticates the server with a TLS certificate. SFTP normally authenticates the SSH host key. Ignoring either identity check makes encryption vulnerable to connecting to the wrong machine.

Passive port ranges, `PASV`, `EPSV`, and `PROT P` belong to FTPS. SSH keys, host-key fingerprints, and subsystem configuration belong to SFTP. Port 990 may be used for implicit FTPS, while port 22 commonly serves SSH/SFTP; neither port is proof of the protocol.

Operational requirements can decide the choice. A partner may require FTPS for an established gateway. A server team may prefer SFTP because one SSH connection is simpler through firewalls.

Take a client configuration and classify every setting as FTP/FTPS, SFTP/SSH, or shared. Remove any setting copied from the wrong protocol.

Check your understanding: FTP security

Check cleartext exposure, anonymous access, bounce attacks, AUTH TLS, protected data channels, certificates, FTPS, and SFTP.

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

Take this interactive quiz online

Use and troubleshoot FTP

Use modern clients, diagnose common failures, automate carefully, and choose a better alternative when appropriate.

Inspect FTP with curl

Use curl verbose output and explicit TLS requirements to observe a transfer without relying on a graphical client.

curl can show the control conversation while handling data connections for you. Start with a server you own or are authorized to test.

```
curl --verbose --user "$FTP_USER" --ssl-reqd \
  ftp://ftp.example.com/path/file.txt --output file.txt
```

`--ssl-reqd` refuses a cleartext FTP transfer. curl prompts for the password when it is omitted. For automation, use a protected credential file or secret store rather than shell history, scripts, logs, URLs, or command arguments.

Verbose output separates control lines:

```
< 220 FTP server ready
> AUTH TLS
< 234 AUTH TLS successful
> EPSV
< 229 Entering Extended Passive Mode (|||50021|)
> RETR path/file.txt
< 150 Opening data connection
```

```
< 226 Transfer complete
```

Lines beginning with `>` go to the server. Lines beginning with `<` came from it. TLS and connection diagnostics appear around them.

Use `--ftp-pasv` or the default passive behavior for ordinary networks. `--ftp-port` selects active mode and needs a reachable client endpoint. Do not change modes randomly; match the failure to the data connection direction.

For an upload, use `--upload-file local.txt` and a temporary remote URL. Check curl's exit status and the FTP completion reply before a later rename or publication step.

Run a download with `--verbose` against an authorized server. Redact credentials and session identifiers, then annotate the greeting, TLS upgrade, data endpoint, preliminary reply, and completion reply.

Configure an FTP client deliberately

Choose the exact protocol, TLS policy, passive mode, transfer type, and certificate behavior instead of accepting vague defaults.

Start by selecting FTP, explicit FTPS, or SFTP correctly. Then configure host, port, username, authentication, and certificate or host-key verification.

For FTP or FTPS behind ordinary networks, choose passive or extended passive mode. Use binary transfers unless you have a specific legacy text-conversion requirement.

Test a listing, a small upload to a temporary name, a download, and a rename. Record the final replies so the configuration can be reproduced outside the GUI.

Write the connection profile as explicit facts:

```
protocol: explicit FTPS
host: files.partner.test
port: 21
TLS: required
certificate validation: required
data mode: EPSV/passive
transfer type: binary
```

"FTP" and "encryption if available" are not precise enough. Opportunistic fallback can expose credentials. Require the agreed protocol and fail closed.

Use a narrow account rooted at the required directory. Prefer a secret manager or protected credential store. For SFTP, pin or verify the SSH host key through a trusted channel; for FTPS, validate the certificate hostname and chain.

Timeouts and retries need command awareness. Retrying a `RETR` into a temporary file is safer than retrying `APPE`, `DELE`, or a rename without first checking remote state.

Reproduce the GUI profile with a command-line client. If the two behave differently, compare the exact TLS mode, passive endpoint, proxy settings, and certificate handling.

Troubleshoot common FTP failures

Classify failures by control, authentication, data, TLS, or filesystem layer before changing settings.

No greeting suggests DNS, routing, port, or service trouble. `530` points to authentication or account policy. Login followed by a hanging listing points to the data connection.

A TLS certificate error is not repaired by disabling validation. Fix the hostname, certificate chain, clock, or server configuration.

`550` commonly means the requested file action is unavailable: inspect the remote path, working directory, permissions, quotas, and server logs. Preserve the exact numeric reply in your diagnosis.

Work from the outside inward:

1. Resolve the hostname and reach the configured TCP port.
2. Read the `220` greeting.
3. Complete TLS and validate identity when required.
4. Authenticate and reach `230`.
5. Confirm `PWD`, transfer type, and features.
6. Negotiate a data endpoint.
7. Open the data connection.
8. Observe `150`, bytes, and final `226`.

A `425` points at step 6 or 7. A `426` means the data connection started and then failed. A `550` after `RETR` points at path or authorization, while a `552` during upload can point at storage allocation.

Compare client and server clocks for certificate and timestamp confusion. Compare the control peer, passive advertised address, and firewall logs for NAT problems. Keep FTPS data-channel TLS errors separate from ordinary TCP failures.

Do not disable certificate validation, open every firewall port, or grant broad filesystem permissions as diagnostic shortcuts. Those changes hide the first failing layer and create a security problem.

Take one real verbose trace and write a one-sentence diagnosis naming the first failed layer, exact reply, and next narrow test.

When FTP still fits

Choose FTP, FTPS, SFTP, HTTPS, or object storage according to interoperability, security, automation, and publishing needs.

Use plain FTP only inside a deliberately protected legacy environment when replacement is not yet possible. Prefer FTPS when a partner mandates the FTP protocol but supports TLS.

Prefer SFTP when you control SSH-based server access and want one encrypted connection. Prefer HTTPS uploads or object-storage APIs for modern web applications, signed links, fine-grained credentials, and cloud workflows.

For a final exercise, draw one required transfer. Mark trust boundaries, protocol, ports, authentication, encryption, data direction, retry behavior, integrity check, and how partial files stay hidden. The diagram should justify the protocol instead of choosing by habit.

Use a decision table:

Constraint	Strong default
Existing partner requires FTP semantics	Explicit FTPS with protected data channels
You manage SSH accounts on both ends	SFTP
Browser or service uploads	HTTPS API or signed upload URL

Constraint	Strong default
Large cloud objects and lifecycle rules	Object storage API
Unreplaceable cleartext appliance	Isolated network, narrow gateway, replacement plan

Also define failure semantics. Can an upload resume? Is a retry idempotent? How do consumers avoid partial files? Which hash or application check proves integrity? Who rotates credentials and reviews access?

Compatibility is a reason to keep FTP, not a reason to ignore its risks. Put a protected gateway near the legacy endpoint instead of spreading cleartext credentials and wide passive ranges across the network.

Revise the transfer diagram with one deliberate network failure after data EOF but before final confirmation. Explain how the automation discovers whether it is safe to retry.

Check your understanding: practical FTP

Check curl usage, client configuration, layered troubleshooting, exact replies, credential handling, and choosing modern transfer alternatives.

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/ftp/.