VTX Transfer 1: file transfer

Nov 15, 2026 · VTX 1 · extension · draft d02 · MIT licence · vtx.foundation/spec/v1/transfer-1/

Transfer 1 moves files between client and server on their own channels: one file at a time or whole sets of files and folders, downloads the user saves through a prompt in the trusted zone, uploads the user starts by picking, dropping or pasting, and files copied and pasted between the session and the device. Every file is checked by SHA-256 and resumable after a reconnect.

§1 Status and scope

Transfer 1 is draft d02 of an extension of VTX 1, in conformance profile 1, built on the core at draft d03. It fills the core's extension points, adds one dynamic channel type, `transfer`, and two frames on the core's clipboard channel; no rule of the core changes. Draft d02 adds sets of files and folders (Section 7) and files on the clipboard (Section 8). It adds:

- one `transfer` channel per file or per set of files and folders, opened by whoever sends them, carrying them under v1 flow control;
- downloads that the user saves through a prompt in the trusted zone, never written without consent;
- uploads that start only from a user gesture: a `files` field, a drop target, or a paste;
- SHA-256 checks on every file, and resume from where a broken transfer stopped;
- files on the clipboard in both directions, as people expect from an RDP session: listed when copied, sent only when pasted;
- application lines, so an agent can hand the user a report or receive their logs;
- ZMODEM and iTerm2 file downloads in the VT engine, so `sz` and `rz` work in a terminal.

Not in Transfer 1. Drive redirection, where the server browses and changes a client folder while it is in use, is Drives 1: the user shares a folder, and the server lists and changes only what lies inside it.

§2 Capabilities and limits

A client says what it can do under the `transfer` key that v1 reserved, and three limits bound what a server can make it store.

Key	Value	Meaning
`transfer`	`download`, `upload`, `folders`, `sets`, `clipboard`, each `true` or absent	Saves files the user accepts; sends files the user picks, drops or pastes; sends one file with its folders as a path; sends and receives sets of files and folders (Section 7); copies and pastes files through the clipboard, which needs `sets` too (Section 8)

Limit	Default	Bounds
`transfer_bytes`	None	The largest file, or set in total, the client accepts; past it the client closes the channel with `limit_exceeded`
`transfer_channels`	4	Transfer channels open at once

Limit	Default	Bounds
`transfer_files`	10,000	Entries, files and folders together, in one set the client receives

The server's own limits for uploads live in the scene, on the files field or drop target (Section 5), so the user sees them before picking anything.

§3 The transfer channel

One `transfer` channel carries one file (OFFER) or one set of files and folders (SET), and the side that sends opens it: the server for a download, with an even ID from 16, and the client for an upload, with an odd one. The only OPEN param is `dir`, `down` or `up`.

- The peer answers OPEN_OK at once. The user's decision comes later, on the channel itself, so either side can withdraw at any moment with CLOSE.
- Data rides the channel's stream under v1 credits or QUIC flow control, never datagrams. Transfers stay last in the priority order with the other dynamic channels, so they never slow input or the scene.

Type	Frame	Sent by	Fields	Purpose
0x10	OFFER	Sender	`id`, `name`, `path`, `size`, `mime`, `sha256`, `mtime`, `note`, `target`, `at`, `source`	Describes one file; the first frame, unless SET is
0x15	SET	Sender	`id`, `name`, `entries`, `note`, `target`, `at`, `source`	Lists a set of files and folders instead (Section 7)
0x11	ACCEPT	Receiver	`offset`; in a set also `entry`, `skip`	Go ahead, from this byte: 0 unless resuming; in a set, the files the receiver does not want
0x12	DATA	Sender	`offset`, `bytes`; in a set also `entry`	The next piece, starting where the last one ended
0x13	END	Sender	`size`, `sha256`; in a set also `entry`	A file is sent: its total and its hash
0x14	DONE	Receiver	`ok`, `reason` (`hash`, `size`, `storage`, `policy`); in a set also `failed`	Everything passed its checks and was kept, or which files did not and why

Endings. Each ends with CLOSE on the channel:

- after DONE, either side sends CLOSE `no_error`;
- when the user declines or cancels, CLOSE `user_stopped` (0x0E, the code Media 1 assigned);
- when a file breaks a limit, such as a field's `max`, CLOSE `limit_exceeded` (0x0A);
- when the receiver will not take a name or type, CLOSE `policy_violation` (0x0C);
- when the sender withdraws before END, CLOSE `no_error` from the sender.

§4 Downloads

A download reaches the disk only after the user says yes in the trusted zone.

- When OFFER arrives, the client shows a save prompt in the trusted zone: the file's name, size and type, the server's verified identity, and the server's `note`. The server cannot draw over the prompt or style it.

- Saving, to a folder the user picks or the usual downloads folder, sends ACCEPT. Declining sends CLOSE `user_stopped`.
- Executables and scripts get an explicit warning in the prompt.
- The client `MAY` offer to always save a type of file from this server. That grant is kept per server identity, and the user can revoke it, as with a v1 permission of scope `always`.
- The client writes to a temporary file, checks size and SHA-256 at END, then moves the file into place and sends DONE. It never opens or runs a downloaded file, and marks it with its origin where the platform has such a mark: Mark of the Web on Windows, the quarantine attribute on macOS.

  Diagram: download flow · offer, prompt, data, check. Drawn in the PDF version, `05-transfer-1.pdf`.

The channel opens at once, but no byte is sent until the user saves, and the file is kept only once its size and hash match.

§5 Uploads

An upload starts only from something the user does: picking files in a `files` field, or dropping or pasting them on a drop target. The server can neither open a file picker nor read a file on its own.

Files field.

```
{"op":"open","id":"logs","type":"files","parent":"intake","label":"Attach
logs","accept":[".log","text/plain"],"multiple":true,"max_files":10,"max":52428800}
```

Prop	Meaning
`label`	Required
`accept`	Media types (`image/png`, `image/*`) or extensions (`.pdf`); the picker offers only these
`multiple`	More than one file; needed for `max_files` above 1 and for `folders`
`folders`	Whole folders, sent as one set (Section 7)
`max`	The largest file, in bytes
`max_files`	The most files
`required`	The form needs at least one file

The client uploads a pick as soon as it is made, one file with OFFER and several files or a folder as one set, and shows the progress in the field. In the form's submit intent, the field's value is the list of upload IDs, so the application can match it to the files it received.

Drop targets. `drop`, with the same members except `label` and `required`, makes a region, a grid or any surface take dropped or pasted files. OFFER `at` gives the drop position in the target's own coordinates: view units on a vector surface, stream pixels on a pixel surface, `[col, row]` on a grid. A file dropped on a remote desktop therefore lands where the user let go.

- The client checks `accept`, `max` and `max_files` before it opens a channel; the server checks them again and refuses a file outside them with `policy_violation` or `limit_exceeded`.

- A folder drop is confirmed in the trusted zone, with the number of files and the total size, before anything is sent.
- Symbolic links, devices and other special files are never sent.

§6 Names, integrity and resume

Names are checked at both ends, every file is checked by size and SHA-256, and a broken transfer picks up where it stopped.

Names.

- `name` is one path segment: 1–255 bytes of UTF-8, no `/` or `\`, no control characters, and not `.` or `..`. An OFFER puts the folders above a lone file in `path`, outermost first, and a set entry's path holds its folders and its name; every segment follows the same rules.
- A receiver still treats every name as untrusted. It changes what its file system cannot hold, such as Windows's reserved names `CON` and `NUL` or trailing dots and spaces, never writes outside the folder the user chose, and never overwrites a file without asking.

Integrity.

- END carries the total size and the SHA-256. The receiver compares both with what it received, and with OFFER's `size` and `sha256` when OFFER gave them; DONE reports the outcome. After a failed check (`ok: false`, reason `hash` or `size`) the file is discarded.
- When the size is not known in advance, as for a log still being written, OFFER leaves `size` out and END gives it.

Resume.

- A detach closes every transfer channel. After the resume, the sender offers the file again, with the same `id`, on a new channel.
- A receiver that kept the partial file answers ACCEPT with the `offset` it has, and DATA continues from there. Resuming needs `sha256` in OFFER, so the receiver knows it is the same file.
- The user is not asked again about a download already accepted.
- In the example trace `trace-download-quic.jsonl`, a 6,000-byte capture resumes at byte 4,000 after a reconnect, and the hash of the whole file still checks out.

§7 Sets of files and folders

A set sends many files and folders as one transfer: a dropped folder, a pick of thirty files, a report with its charts. The user answers one prompt for the whole set, and empty folders and the executable bit survive the trip. The sender's first frame is SET instead of OFFER.

```
{"id": "s1", "name": "report-2026", "note": "Quarterly report with its charts",
"entries": [
  {"path": ["report-2026"], "dir": true},
  {"path": ["report-2026", "summary.md"], "size": 900, "mime": "text/markdown",
"sha256": "…"},
  {"path": ["report-2026", "charts"], "dir": true},
  {"path": ["report-2026", "charts", "q3.csv"], "size": 2400, "mime": "text/csv",
"sha256": "…"},
  {"path": ["report-2026", "render.sh"], "size": 300, "sha256": "…", "exec": true}]}
```

Entries.

- Each entry has a `path` relative to the set, every segment following the name rules of Section 6. No two entries share a path.
- A folder (`dir: true`) has no `size`, `mime`, `sha256` or `exec`. A file has a `size`; `sha256` is needed to resume it, and `exec` marks it executable.
- A folder comes before anything in it, so the receiver can create folders first. Empty folders are entries like any other.
- Links are not entries: the sender sends what a link points to, or leaves it out.
- A download may hold at most `transfer_files` entries and `transfer_bytes` in total.

The exchange.

1. The receiver answers ACCEPT once for the whole set. `skip` lists files it does not want, for example a file it already has with the same `sha256`, or one the user unticked.
2. The sender sends each remaining file in the order of the list: DATA frames with `entry`, then END with `entry`, `size` and `sha256`. An empty file is just its END.
3. After the last END, the receiver answers DONE. `ok` is true only if every file passed its check; `failed` lists the files that did not, each with its reason.

Resume. A set keeps its `id` across a reconnect. When the sender offers the same set again, the receiver's ACCEPT names the `entry` to continue from and the `offset` within it. Every file before that entry must have arrived whole.

Prompts and targets. A download of a set shows one prompt in the trusted zone: the set's `name`, how many files and folders, the total size, and where to save it. An upload of a set needs a target that accepts it: `folders` for any folder, `multiple` and `max_files` for the number of files, and `accept` and `max` for each file. A file sent alone still uses OFFER, with `path` if its folders matter.

§8 Files on the clipboard

Copy and paste with files works in both directions, as people expect from an RDP session, and for folders as well as files. Copying a folder copies everything in it: subfolders, empty folders and files, with their names, sizes and executable bit. Copying only lists the files. Their bytes travel when someone pastes, as a set.

A clipboard item of type `application/vnd.vtx.files+cbor` carries the list: an `id` and `entries`, exactly as in a set (Section 7). It rides v1's own clipboard frames, and two frames join channel 5:

Type	Frame	Sent by	Fields	Purpose
0x14	FETCH	Either side	`id`, `entries`	Send me these files from your clipboard; absent `entries` means all of them
0x15	GONE	Either side	`id`	Those files are no longer on my clipboard

Copy in the session, paste on the device. The server writes the list to the client's clipboard with WRITE, under v1's rules for clipboard writes. The client puts a promise of those files on the device's clipboard, without fetching anything. When the user pastes, in a file manager or a folder, the client sends FETCH, and the server opens a download channel with a SET of the same `id`, `source: "clipboard"`, and exactly the entries fetched. The paste is the user's consent and chooses the place, so there is no second prompt; the client still checks every name and limit.

Copy on the device, paste in the session. Pasting into a VTX application needs nothing new: the client uploads a set with `source: "paste"` to the field or drop target. Pasting into a remote desktop does: the remote application asks the remote clipboard, so the server first needs the list. It sends READ with the files type. The client includes a list only once the user has granted the `clipboard.files` permission in the trusted zone (scope `once`, `session` or `always`), next to v1's `clipboard.read`. When the remote application actually pastes, the server sends FETCH, and the client uploads the set.

Rules.

- FETCH names a list that is on the other side's clipboard now. A subset keeps every folder above the files it names.
- The SET that answers a FETCH carries the same `id` and exactly the entries fetched: never a file that was not copied.
- Clipboard sets have no `target`: the files go where the user pastes them.
- When the clipboard changes, its owner sends GONE, and the list can no longer be fetched.

Diagram: files on the clipboard · a folder copied each way, 4 parties, 15 messages. Drawn in the PDF version, `05-transfer-1.pdf`.

In both directions the copy moves only the list, and FETCH is the moment the bytes are asked for: by the client when the user pastes on the device, by the server when the remote application pastes. Each folder arrives whole, subfolders and empty folders included.

On a plain v1 client without `clipboard` the server sees no files item and falls back to text, as v1 already does.

§9 Application interface

An application offers downloads and receives uploads in JSON lines; the server runs the channels, the prompts and the checks. `vtx-app-transfer-1.schema.json` and `vtx-app-events-transfer-1.schema.json` define every line.

From the application.

Line	Fields
`download`	`id`, `name`, `mime`, `size`, `sha256`, `mtime`, `note`, and the data: `file` (a path on the server), `base64` (a small file) or neither, when chunks follow
`download_set`	`id`, `name`, `note`, `entries` as in Section 7; each file entry has its own `file`, or chunks with its `entry` follow
`download_chunk`	`id`, `offset`, `base64`; in a set also `entry`
`download_end`	`id` after the last chunk; in a set, `entry` after that entry's last chunk
`download_cancel`	`id`
`clipboard_write`	v1's line; an item of type `application/vnd.vtx.files+cbor` holds `files`: `id` and `entries`, each file with its `file` on the server
`clipboard_fetch`	`id`, `entries`: have the client send files it listed in `clipboard_data`

To the application.

Line	Fields
`download_state`	`id`, `state` (`offered`, `accepted`, `declined`, `done`, `failed` or `cancelled`), `reason`, `bytes`
`upload`	`id`, `target`, `name`, `path`, `size`, `mime`, `source`, `at`, `state` (`started`, `done`, `failed` or `cancelled`), `sha256`, `file`, `reason`; for a file in a set also `set` and `entry`. `target` is absent only for files pasted from the clipboard
`upload_set`	`id`, `state`, `target`, `name`, `source` (`pick`, `drop`, `paste` or `clipboard`), `at`, `entries` when started, `failed`, `reason`; each file then arrives as an `upload` with `set`
`upload_chunk`	`id`, `offset`, `base64`
`clipboard_data`	v1's line; an item of the files type holds `files`: the `id` and `entries` the user copied, without their bytes

- `file` paths work only for an application on the server's own machine, over stdio. There, `upload` with state `done` names where the server stored the file. Over WebSocket, the server sends the bytes as `upload_chunk` lines before `done`.
- `sha256` is hex here and raw bytes on the wire.
- The server checks an inline or chunked download against its `size` and `sha256` before offering it.

An application hands over a build log and receives one back (→ the application writes, ← the server writes):

```
→ {"kind":"download","id":"d1","name":"build-07.log","mime":"text/
plain;charset=utf-8","size":329,"sha256":"a31ed2dd…","base64":"MjAyNi0…"}
← {"kind":"download_state","id":"d1","state":"accepted"}
← {"kind":"download_state","id":"d1","state":"done","bytes":329}
←
{"kind":"upload","id":"u1","target":"logs","name":"build-07.log","size":329,"mime":"text/
plain","source":"pick","state":"started"}
← {"kind":"upload_chunk","id":"u1","offset":0,"base64":"MjAyNi0…"}
←
{"kind":"upload","id":"u1","target":"logs","name":"build-07.log","state":"done","sha256":"a31ed2dd…"}
```

§10 Terminals

Terminal users already move files with ZMODEM and with iTerm2's inline files. The VT engine turns both into Transfer 1, so they work on every client and keep the save prompt.

- **`sz file`.** The engine answers the ZMODEM handshake itself, receives the file from the program and offers it to the client as a download. Nothing reaches the grid.
- **`rz`.** The engine opens a files field over the grid saying that `rz` is waiting. The user's click opens the picker, and the engine feeds the chosen files to `rz` over ZMODEM. Cancelling aborts `rz`.
- **iTerm2 `OSC 1337 ; File=`** with `inline=0` becomes a download offer, its `name` decoded from base64. With `inline=1` it stays an image placement under Surfaces 1.
- When the client lacks the matching `transfer` capability, the engine aborts the ZMODEM session cleanly, so the program does not hang.

§11 Fallbacks and security

No download is written without the user's yes, no upload starts without the user's gesture, and the server only ever learns file names, never local paths.

The client lacks	What happens
`download`	No download is offered; the application hears `download_state failed`, reason `unsupported`
`upload`	Files fields and drop targets show as disabled, with their label
`folders`	The picker and drops take files only
All of it (serial links, grid-only clients)	Files fields show their label as text; the VT engine aborts ZMODEM

Security.

- Names are untrusted at both ends (Section 6).
- The client never opens or runs a downloaded file, and gives it the platform's mark of origin.
- `transfer_bytes` and `transfer_channels` bound what a server can make the client store, and the client checks its free space before ACCEPT.
- Hashes catch corruption and truncation, not malice: a hostile server can send a harmful file with a correct hash. That is why the prompt shows the verified server identity, and warns about executables.
- A drop on a pixel surface sends the file to the remote machine the user is looking at; a folder drop's confirmation names that machine.
- Transfers ride the binding's TLS or QUIC encryption, like everything else in the session.

§12 Definitions and checks

The definitions live in `ext/transfer-1/` of the bundle, and `tools/validate.py` runs 240 checks on them; every check in the bundle passes.

File	What it holds
`vtx-transfer-1.cddl`	The limits, the capability value, the `clipboard.files` permission, OPEN params, the six frame payloads, set entries, and the clipboard file list with FETCH and GONE
`vtx-ops-transfer-1.schema.json`	Every v1, Surfaces 1 and Media 1 op, plus the files field and `drop`
`vtx-app-transfer-1.schema.json`, `vtx-app-events-transfer-1.schema.json`	The application lines, both directions, sets and clipboard files included
`examples/valid/`	A scene; five wire traces: two downloads over QUIC (one resumed), uploads over TLS, a folder download with a skipped file, a dropped folder resumed mid-file, and folders copied and pasted both ways; three application sessions
`examples/invalid/`	9 schema and 5 scene-rule violations, 53 broken traces and 39 broken application lines, each labelled with the error it must raise
`examples/resources/build-07.log`, `examples/vectors.json`	A 329-byte file and every frame of its download, OFFER to DONE
`tools/gen_ext_transfer.py`, `tools/make_transfer.py`, `tools/make_transfer_sets.py`	Generate the schemas, the vectors, and the draft d02 traces and application examples

The 240 checks:

- 22: the schemas are valid, the CDDL parses, and the scene example passes its schema and the scene rules;
- 4: the v1, Surfaces 1 and Media 1 schemas reject files fields and drop targets;
- 14: each invalid scene line fails its own op's schema or a scene rule (`multiple` for `max_files` and `folders`);
- 8: the vector file matches its hash, each frame recomputes byte for byte and parses back, and the DATA frames rebuild the file;
- 39: the CDDL accepts 22 samples and rejects 16, and v1's CDDL rejects the transfer limits;
- 58: a channel simulator replays all five traces, both resumes included, and catches every one of the 53 broken traces for its labelled reason. For single files: opener and channel IDs, capabilities, frame order and sender, names, targets, offsets and sizes, hashes against the bytes, resume offsets, no datagrams, limits and v1 windows. For sets: names and paths, folders before their contents, folder and file fields, `transfer_files`, skips, the order of entries, sizes, per-file hashes reported in DONE, and resume entries. For the clipboard: fetches of lists that exist, folders above fetched files, exactly the fetched entries, no targets, types the READ asked for, and the `clipboard.files` permission;
- 95: every application line passes its schema, all three sessions pass the application rules, and every broken line or session fails for its labelled reason: hashes of chunked data, set entries and their folders, files sent in chunks or from server files, clipboard lists and fetches, and uploads in a set matched to their entries.

§13 Wire summary

Transfer 1 fills v1's extension points and adds entries to registries that take new entries by specification; it changes no v1 rule.

Registry or extension point	Transfer 1 adds
Dynamic channel types	`transfer`
Frame types on it	0x10 OFFER, 0x11 ACCEPT, 0x12 DATA, 0x13 END, 0x14 DONE, 0x15 SET
Frame types on channel 5 (clipboard)	0x14 FETCH, 0x15 GONE
Clipboard item types	`application/vnd.vtx.files+cbor`: a file list, CBOR in the item's data
Capability keys	`transfer`
Limits (`$$limit-ext`)	`transfer_bytes`, `transfer_channels`, `transfer_files`
Permissions (`$permission-ext`)	`clipboard.files`
Close codes	None new: `user_stopped` (0x0E, from Media 1), `limit_exceeded`, `policy_violation`
Node types	`files`
Node props	`drop` on regions, grids and surfaces
Application interface kinds	From the application: `download`, `download_set`, `download_chunk`, `download_end`, `download_cancel`, `clipboard_fetch`, and files in `clipboard_write`. To it: `download_state`, `upload`, `upload_set`, `upload_chunk`, and files in `clipboard_data`