

TCPP Node

How to run and operate a TCPP node: composing schema-driven requests, exchanging signed frames with other nodes, replying to what arrives, and managing identities, peers and records.

Schema 1.3.0 27 request/response pairs, one shared envelope.	ES384 Every frame is a compact JWS over ECDSA P-384.	One binary App, schemas and examples compiled in. Nothing to install.
---	---	---

CONTENTS

01	What a node is	09	Peers and verification
02	Getting started	10	Identity
03	The window	11	Log
04	Compose	12	Workflows
05	The record library	13	Command line and files
06	Validation	14	HTTP API
07	Inbox and replying	15	The TCPP message
08	Sent	16	Appendix

■ What a node is

A TCPP node is a **single executable**. Start it and it does two things at once: it serves a schema-driven web application in your browser, and it listens for TCPP frames from other nodes over TLS. Everything it needs is compiled into the binary: the 1.3.0 JSON schemas, the shipped example payloads and the browser application itself. Nothing else has to be installed, and it reaches out to nothing on the internet.

The unit of work is a **frame**: one TCPP message, signed by the sending node as a compact JWS (a JWT) and posted to a peer. The node records every frame it sends and every frame it receives in an append-only log next to the binary, with the payload, the raw token, who it came from and whether the signature checked out.

What you can do with it

- Compose any of the 27 TCPP request types from a form generated out of the schema.
- Sign and send it to another node, and watch it arrive there live.
- Answer a received request from a reply form that is pre-filled from the request.
- Keep tuned payloads as named *records* and re-send them later.
- Hold peers' public keys and have every stored frame verified against them.
- Read back the full history: payload, token, JOSE header, transport result.

One binary also covers every arrangement you are likely to need while working on TCPP. The node can:

- Exchange frames with itself, so that a single instance is enough for a complete round trip. With no `--peer` given, its own address is the default target.
- Exchange frames with further nodes on the same host, each started on a port of its own.
- Exchange frames with nodes on other hosts across the network. If nothing arrives, the firewall is the first thing to check.
- Load the shipped example payloads, and save and reload test data of your own.
- Create its own key pair, and fetch peers' public keys from them directly, which comes into its own inside an established PKI.
- Stand in as the counterpart while you develop and test a TCPP implementation of your own.

■ Getting started

What you were given

One executable, and nothing else. There is no installer, no runtime to put in place first and no configuration file to write: put it in a directory of its own and start it. The download is built for one operating system and one processor, so pick the file that matches the machine you are on:

FILE	RUNS ON
<code>tcpp-node-windows-amd64.exe</code>	Windows, Intel or AMD processor (the usual one)
<code>tcpp-node-windows-arm64.exe</code>	Windows on ARM
<code>tcpp-node-linux-amd64</code>	Linux, Intel or AMD
<code>tcpp-node-linux-arm64</code>	Linux on ARM, a Raspberry Pi or an ARM server
<code>tcpp-node-darwin-amd64</code>	macOS, Intel
<code>tcpp-node-darwin-arm64</code>	macOS, Apple Silicon (M1 and later)

They are the same program; only the machine code differs. Each is self-contained, with no libraries to install beside it and nothing to unpack, and none of them reaches out to the internet. Rename the one you downloaded to `tcpp-node` (`tcpp-node.exe` on Windows), which is how this manual writes it throughout.

Give it a directory you can find again. Everything the node creates (its key, its certificate, its frame log, its records) is written next to the executable, so the directory *is* the node.

Run

Start it with no arguments and it listens on port 8443.

WINDOWS

In PowerShell or `cmd`, from the node's directory. Double-clicking it works too, but then the window closes with the node and you never see why it stopped.

```
.\tcpp-node.exe
```

MACOS AND LINUX

A downloaded file is not executable yet, so mark it once:

```
chmod +x tcpp-node
./tcpp-node
```

THE DOWNLOAD IS UNSIGNED

It carries no code-signing certificate, so each system objects once, on the first start only:

- **Windows.** SmartScreen says the publisher is unknown: *More info* → *Run anyway*. If it refuses outright, the file is marked as coming from the internet: *right-click* → *Properties* → *Unblock*, then *OK*.
- **macOS.** Gatekeeper blocks it: open the containing folder, *right-click the file* → *Open* and confirm, or clear the mark with `xattr -d com.apple.quarantine tcpp-node`.
- **Linux.** Nothing objects; `chmod +x` is all it needs.

From there on it is the same program everywhere. The first run creates an identity and prints what it is about to serve:

```
2026/09/03 18:03:48 created C:\Users\User\tcpp-lab\a\tcpp-key.pem
2026/09/03 18:03:48 created C:\Users\User\tcpp-lab\a\tcpp-cert.pem
2026/09/03 18:03:48 TCPP node CSU / P:NODE-9BED6853
2026/09/03 18:03:48 listening https://localhost:8443/
2026/09/03 18:03:48 advertising localhost:8443
2026/09/03 18:03:48 signing ES384, kid 9bed6853
2026/09/03 18:03:48 identity C:\Users\User\tcpp-lab\a
2026/09/03 18:03:48 frames 0 replayed from tcpp-frames.jsonl
2026/09/03 18:03:48 schemas 27 endpoints
2026/09/03 18:03:48 records 63 in C:\Users\User\tcpp-lab\a\records
2026/09/03 18:03:48 target localhost:8444
2026/09/03 18:03:48 The certificate is self-signed, so the browser will
```

Open **https://localhost:8443/**. The certificate the node generated for itself is self-signed, so the browser interrupts once with a privacy warning; continue past it (in Chrome: *Advanced* → *Proceed*). The warning does not come back for that host until the node is given a new certificate.

THE NODE HAS NO LOGIN

Nothing in the HTTP interface is authenticated. Anyone who can reach the port can read the frame log, save records, change peers and send frames as this node. Run it on a network you trust (a laptop, a lab segment, a private VPN), not on a public address.

What appears on disk

All state lives beside the binary (or in `--data-dir`). A node is a directory you can copy.

FILE	CONTENTS	MODE
<code>tcpp-key.pem</code>	ECDSA P-384 private key, PKCS#8; the secret	<code>0600</code>
<code>tcpp-cert.pem</code>	self-signed certificate for the same key, valid 10 years	<code>0644</code>
<code>tcpp-node.json</code>	this node's TCPP type and id	<code>0644</code>
<code>tcpp-frames.jsonl</code>	every frame sent and received, one JSON object per line	<code>0644</code>
<code>tcpp-peers.json</code>	known peers and their public keys	<code>0644</code>
<code>records/</code>	the record library, <code><schema>/<name>.json</code>	<code>0755</code>

The one-node smoke test

You do not need a second node to try the round trip. With no `--peer` the default target is `localhost:<port>`, this node itself, so a frame it signs is delivered to its own inbox. Open **Compose**, pick **ping**, press **Load...** → **example**, then **Sign & Send**. It appears under **Inbox** immediately.

The two-node lab used in this manual

Every screenshot in this manual comes from two nodes on one machine: **A**, a Charge Switch Unit on port 8443, and **B**, a Trusted Management Host on port 8444. Each has its own directory, so each has its own key, log and records.

```
# macOS and Linux: two directories, one copy of the binary in each
mkdir a b
cp tcp-node a/ && cp tcp-node b/

# terminal 1
a/tcp-node --port 8443 --node-type CSU --peer localhost:8444
# terminal 2
b/tcp-node --port 8444 --node-type TMH --peer localhost:8443
```

```
# PowerShell
mkdir a, b
copy tcp-node.exe a\; copy tcp-node.exe b\

# window 1
a\tcp-node.exe --port 8443 --node-type CSU --peer localhost:8444
# window 2
b\tcp-node.exe --port 8444 --node-type TMH --peer localhost:8443
```

Open <https://localhost:8443/> and <https://localhost:8444/> in two browser tabs and you can watch both sides of an exchange at once.

■ The window

The screenshot displays the TCPP Node application interface. At the top, a header bar shows the node name 'TCPP Node', schema version '1.3.0', and connection status 'LIVE'. Below the header, the interface is divided into three main sections:

- Compose:** A sidebar on the left with a 'Compose' button and a list of actions (Inbox, Sent, Peers, Identity, Log) and a search filter.
- REQUEST SCHEMAS:** A central panel showing a list of request schemas (archive get, archive list, charge event, etc.) and a 'ping' schema selected. The 'ping' schema details include fields like 'url to tcpp version infos', 'head', 'record_type', 'record_direction', 'record_udo', 'record_pagination', 'record_version', 'origin_type', 'origin_id', 'target_type', 'target_id', 'context_type', 'context_id', and 'error_code'.
- OUTGOING REQUEST:** A panel on the right showing a JSON representation of the outgoing request, including fields like 'tcpp', 'head', 'record_type', 'record_direction', 'record_udo', 'record_pagination', 'record_version', 'origin_type', 'origin_id', 'target_type', 'target_id', 'context_type', 'context_id', 'error_code', and 'error_text'.

At the bottom right, there is a 'SEND TO' field with 'localhost:8444' and a 'Sign & Send' button.

FIGURE 01 • The whole application. The header identifies the node, the rail on the left switches views, and the three columns belong to **Compose**.

Header

The chip at the top always names the node you are looking at, which matters once two tabs are open:

- **CSU / P:NODE-9BED6853** is the node's TCPP module type and id.
- **:8443** is the port this page was served from.
- **ES384 • 9bed6853** is the signing algorithm and the short key id that appears in every token it signs.

On the right, **LIVE** means the browser is attached to the node's event stream and new frames will appear on their own. It turns into **OFFLINE** when the connection drops.

Navigation rail

VIEW	WHAT IT IS FOR
Compose	Build a request from a schema and send it.
Inbox	Frames that arrived here. The red badge counts unread ones.
Sent	Frames this node sent, with their transport result.
Peers	Known nodes, their addresses and public keys.
Identity	This node's own key, fingerprint and file paths.
Log	Everything, sent and received, as dense lines.

The foot of the rail shows how many frames the node currently holds. Short confirmations (*sent to localhost:8444, peer saved, loaded ping_request/example*) appear briefly as a blue toast in the bottom-right corner.

Light and dark

The application follows your operating system's colour scheme. Nothing else changes.

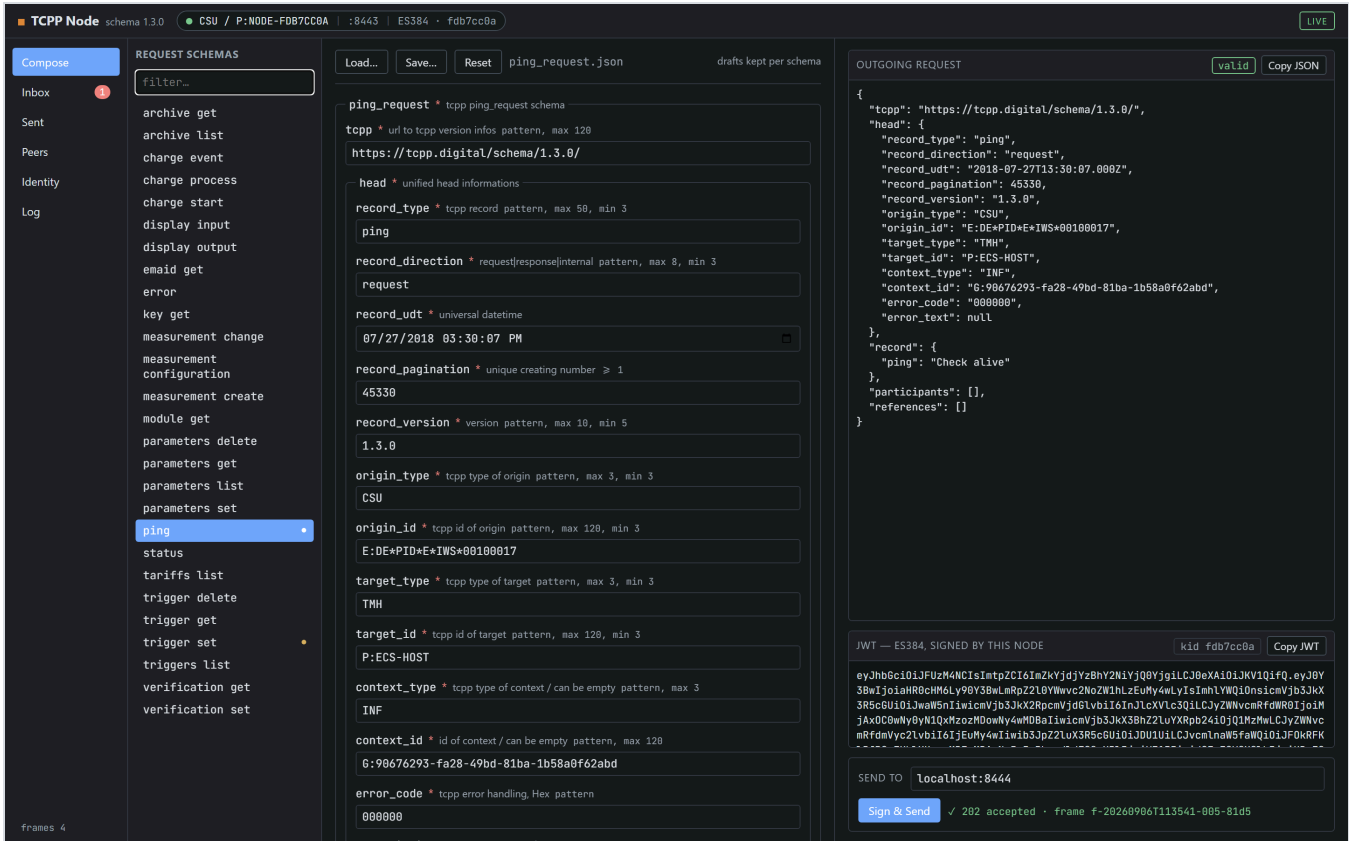


FIGURE 02 · The same screen with the system set to dark.

■ Compose

Compose is three columns: the request schemas on the left, a form generated from the selected schema in the middle, and what will actually go out on the right.

Choosing a schema

The left column lists all 27 request types. Type in **filter...** to narrow it down; the filter matches the schema name, so **charge** leaves the three charge endpoints.

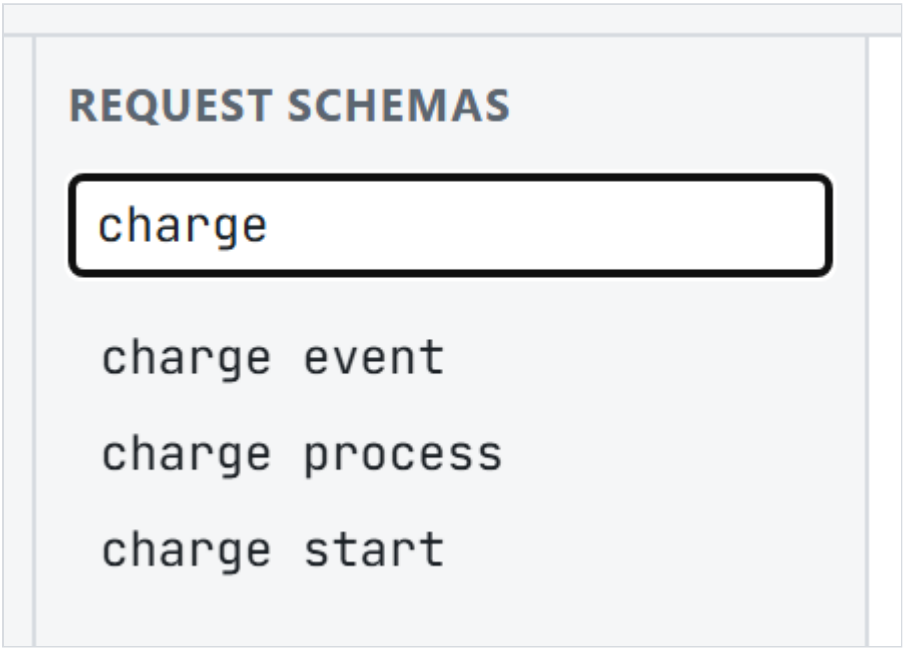


FIGURE 03 • The schema list, filtered. A small amber dot marks a schema you have an unsent draft for; the selected schema is highlighted.

The generated form

The middle column is built from the JSON schema, not hand-written, so it always matches what the node will accept. Each field shows its name in monospace, a red ^{*} when it is required, the schema's description, and a compact summary of its constraints (`pattern`, `max 120`, `min 3`, `≥ 1`).

SCHEMA SAYS	YOU GET
<code>enum</code>	a dropdown, with a <code>(none)</code> entry when the field may be omitted or null
<code>format: date</code>	a date picker
<code>format: date-time</code>	a date-and-time picker, read back as an ISO timestamp

SCHEMA SAYS	YOU GET
string, long <code>maxLength</code>	a text area
string	a text box; hover it to see the pattern it must match
number, integer	a number box carrying <code>minimum</code> and <code>multipleOf</code>
boolean	a checkbox
object	a bordered group; a nullable one gets a <i>set to null</i> switch that hides its body
array	numbered item cards with <code>Remove</code> , and a <code>+ Add ...</code> button underneath
object without properties	a raw JSON text area, validated as you type

record

trigger_check_sign * CLOCK.TIME[CHECK.HARDWARE]...

pattern, max 50, min 5

CLOCK.TIME

trigger_steps f.e. 'CLOCK.TIME': 5 = Every 5 Minutes

15

subject_id optional id of legal_subject pattern, max 120

module_id optional id of module pattern, max 120

☒ pause_on_breakNot try while break/deactivate/shutdown/etc...

☐ value_was_changedOnly send if value was changed

value_min if value number/string/boolean arrived a minimum

value_max if value number/string/boolean arrived a maximum

value_exact if value number/string/boolean arrived a exact value

+ Add trigger_formula

trigger_actions * List of actions start

#1

Remove

record

action_sign * ACTION.REQUEST[... pattern, max 50, min 5

ACTION.REQUEST

action_id * id of action pattern, max 120

G:eb735807-d849-4d98-b8d1-e36f7ae62164

target_record_type tcp record pattern, max 50

status

target_type request to target type pattern, max 3

CPD

FIGURE 04 · `trigger_set` scrolled down to `record`: checkboxes, a numbered array item with `Remove`, and `+ Add trigger_formula` to append another.

What goes out

The right column mirrors the form as you type. **Outgoing request** is the exact JSON that will be signed, and the badge beside it says `valid` or how many issues the schema found. The **JWT** panel is empty until you send: the node signs, so the

private key never enters the browser. After a send it holds the compact token, and [Copy JSON](#) / [Copy JWT](#) put either on the clipboard.

Send to takes a `host:port`. It starts out holding the node's default target (`--peer`) and offers your known peers as suggestions; a bare host or IP gets this node's own port appended. Press [Sign & Send](#) and the status line beside it reports the outcome.

DRAFTS

Each schema keeps its own draft while the tab is open, so you can jump to another schema and come back to what you were typing. [Reset](#) throws the draft away and rebuilds the form empty. Drafts do not survive a page reload; to keep something, save it as a record.

■ The record library

A record is a stored payload on the node, at `records/<schema>/<name>.json`. On first start the node writes every example shipped in the binary into that directory, and from then on the library is yours to grow. `Load...` and `Save...` sit in the toolbar of both the compose form and the reply form.

The screenshot shows the TCPP Node interface for schema 1.3.0. The left sidebar lists various actions like 'Compose', 'Inbox', 'Sent', 'Peers', 'Identity', and 'Log'. The main area is divided into 'REQUEST SCHEMAS' and 'OUTGOING REQUEST'. The 'REQUEST SCHEMAS' section shows a list of schemas, with 'ping_request' selected. Below this, a table lists records for 'ping_request - records'. The table has columns for 'example', 'shipped', 'date', and 'action'. Two records are shown: 'example' (shipped) and 'lab ping' (saved). The 'lab ping' record is highlighted. Below the table, the 'head' section shows fields for 'record_type', 'record_direction', 'record_udt', 'record_pagination', 'record_version', 'origin_type', 'origin_id', 'target_type', 'target_id', 'context_type', 'context_id', and 'error_code'. The 'OUTGOING REQUEST' section shows a JSON payload for a 'ping' request, including fields like 'tcpip', 'head', 'record_type', 'record_direction', 'record_udt', 'record_pagination', 'record_version', 'origin_type', 'origin_id', 'target_type', 'target_id', 'context_type', 'context_id', 'error_code', and 'error_text'. The 'SEND TO' field is set to 'localhost:8444'.

example	shipped	date	action
example	shipped	2026-09-06 11:34	x
lab ping	saved	2026-09-06 11:35	x

```
{  "tcpip": "https://tcpip.digital/schema/1.3.0/",  "head": {    "record_type": "ping",    "record_direction": "request",    "record_udt": "2018-07-27T13:30:07.000Z",    "record_pagination": 45330,    "record_version": "1.3.0",    "origin_type": "CSU",    "origin_id": "E:DE*PID+E:IWS*00100017",    "target_type": "TMH",    "target_id": "P:ECS-HOST",    "context_type": "INF",    "context_id": "6:90676293-fa28-49bd-81ba-1b58a0f62abd",    "error_code": "000000",    "error_text": null  },  "record": {    "ping": "Check alive"  },  "participants": [],  "references": []}
```

FIGURE 06 • `Load...` lists the records stored for this schema. A `shipped` record came with the binary, a `saved` one is yours. `x` deletes one, after a confirmation.

The screenshot shows the TCP Node interface with a 'Save...' dialog box open. The dialog has a title bar 'Save as - ping_request' and a text input field containing 'lab ping'. Below the input field is a 'Save' button. The background interface is divided into three main sections:

- REQUEST SCHEMAS:** A list of schemas on the left, with 'ping' selected. The main area shows the details for the 'ping' schema, including fields like 'record_type', 'record_direction', 'record_udt', 'record_pagination', 'record_version', 'origin_type', 'origin_id', 'target_type', 'target_id', 'context_type', 'context_id', and 'error_code'.
- OUTGOING REQUEST:** A panel on the right showing a JSON payload for a ping request. The payload includes fields like 'tcp', 'head', 'record', 'participants', and 'references'.
- JWT:** A section at the bottom right showing a JWT token and a 'Sign & Send' button.

FIGURE 07 • **Save...** asks for a name. It offers the name you last loaded, warns when you are about to overwrite, and refuses names outside `letters, digits, space, dot, underscore, dash`.

Loading a record replaces the form contents; saving stores what the form currently holds. The bytes are kept **exactly** as they were, key order included, because those bytes are what gets signed the next time the record is loaded and sent.

RESTORING THE SHIPPED EXAMPLES

Seeding happens per schema directory: one that already exists is left alone, so a record you delete stays deleted. Delete `records/<schema>/`, or all of `records/`, and the shipped examples come back on the next start. The embedded examples are read-only and always available at `/api/example/...`, so editing a record can never damage them.

Validation

The form is checked against the schema as you type. A field that fails gets a red border and the reason underneath it, every group containing a bad field is outlined, and the badge above the JSON pane counts what is left. Click the badge to jump to the first offending field.

The screenshot shows the TCPP Node interface with the following components:

- Compose** button and a list of actions (Inbox, Sent, Peers, Identity, Log, etc.).
- REQUEST SCHEMAS** section with a search filter and a list of schemas (archive get, archive list, charge event, etc.).
- Schema Editor** for 'ping_request' schema, showing fields like 'tcpip', 'head', 'record_type', 'record_direction', 'record_ult', 'record_pagination', 'record_version', 'origin_type', 'origin_id', 'target_type', 'target_id', 'context_type', 'context_id', and 'error_code'. The 'not-an-id' field is highlighted with a red border and a message: "does not match pattern ^https?://(?:www\\.|)?[a-zA-Z0-9@%\\.\\+~#=[\\]256\\a-zA-Z0-90]{1,63}\\b(?:[a-zA-Z0-9@%\\.\\+~#&\\/=]*)\$".
- OUTGOING REQUEST** section with a '1 issue' badge and a 'Copy JSON' button. The JSON output is shown, including the 'tcpip' field with the value 'not-an-id'.
- JWT** section with a 'Copy JWT' button and a 'Sign & Send' button.

FIGURE 08 • One field fails its pattern. The message under the box is the actual rule from the schema, and the badge reads **1 issue**.

The field you are currently typing in is never marked; the marks are applied when you leave it, so the form does not flash red while you work.

ISSUES NEVER BLOCK A SEND

Sign & Send stays enabled no matter how many issues there are. Sending a deliberately malformed record is a legitimate way to test what a peer does with one, and the peer, not the sender, is the authority on what it accepts.

Checked keywords are `type`, `required`, `properties`, `items`, `minItems`, `minLength`, `maxLength`, `pattern`, `minimum`, `maximum`, `multipleOf`, `enum`,

`format: date`, `format: date-time` and local `$ref`. Anything else in the schema is ignored rather than guessed at.

■ Inbox and replying

Everything another node sends here lands in **Inbox**, including frames that could not be parsed, which are recorded rather than dropped. New arrivals appear without a reload and raise a toast.

TCPP Node schema 1.3.0

● TMH / P:NODE-DA7083FF | :8444 | ES384 · da7083ff

Compose

Inbox 3

Sent

Peers

Identity

Log

Received from any

any 1 peer 3

CSU-node localhost:8443 3 · 3 new

Frames 3 frames

Mark all read

● ping REQUEST 13:35:41

from CSU-node signature ✓

● measurement_create REQUEST 13:35:23

from CSU-node signature ✓

● charge_start REQUEST 13:35:23

from CSU-node signature ✓

FIGURE 09 · The left column of the inbox. The upper pane filters by sender; the lower one lists frames, newest first, with **Mark all read** on the right.

Reading a frame line

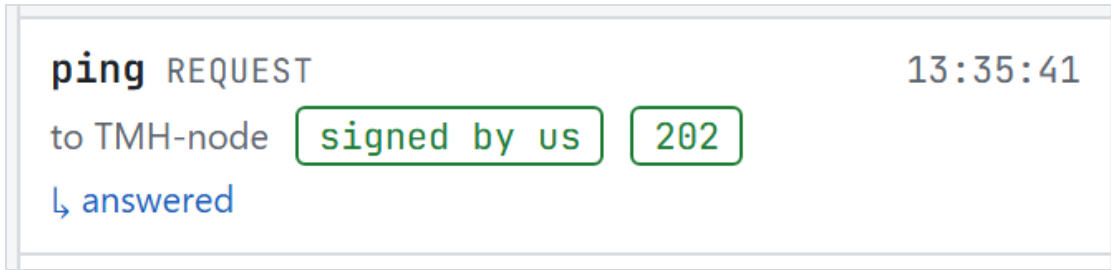


FIGURE 10 · One entry. Top: the record type, its direction and the time. Middle: the peer and the state badges. Bottom: how it relates to another frame.

The third line appears only when the frame is part of a pair:

- ↳ answered : a request of yours that has since been answered.
- ↳ awaiting response : a request that has not been answered yet.
- ↳ answers an earlier request : this frame is the answer to one.

Frame detail

Selecting a frame opens it on the right and marks it read.

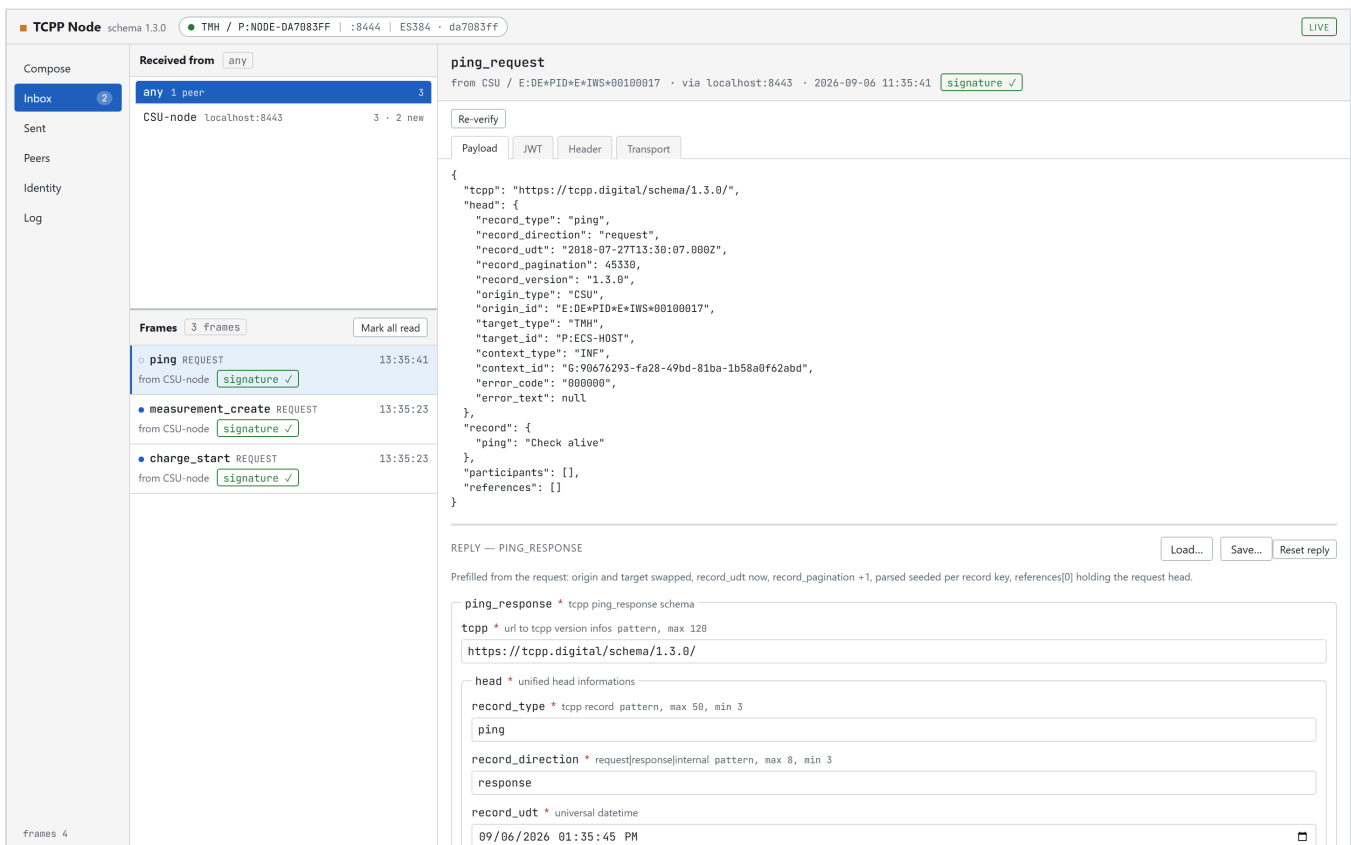


FIGURE 11 · A received request. The line under the title names who sent it, over which address it arrived, when, and how its signature checked out.

Four tabs hold everything the node kept:

TAB	SHOWS
Payload	the decoded TCPP message, formatted.
JWT	a one-line summary and the raw compact token as it arrived.
Header	the decoded JOSE header: <code>alg</code> , <code>kid</code> , <code>typ</code> .
Transport	frame id, direction, peer, verification state, delivery result, correlation links.

[illegible]

FIGURE 12 • The JWT tab. The token is exactly the bytes that were verified; nothing was re-encoded on the way in.

Re-verify checks the stored signature again against whatever key the peer has now, and reports the result in a toast. It is the button to press after adding or changing a peer's key.

Replying

When the open frame is a *request* and the schema set has a matching response, a reply form is appended below the tabs, already filled in from the request:

- origin and target swapped, since you are answering the sender;
- `record_udt` set to now and `record_pagination` raised by one;
- `error_code` reset to `000000` rather than echoing the request's;
- `parsed` seeded with an entry per record key;
- `references[0]` holding the request's `head`, which is what lets both nodes pair the two frames.

TCPP Node schema 1.3.0 | TMH / P:NODE-DA7083FF | :8444 | ES384 - da7083ff | LIVE

Compose

Inbox 2

Sent

Peers

Identity

Log

Received from any

any 1 peer 3

CSU-node localhost:8443 3 · 2 new

Frames 3 frames Mark all read

ping REQUEST 13:35:41
from CSU-node signature ✓

measurement_create REQUEST 13:35:23
from CSU-node signature ✓

charge_start REQUEST 13:35:23
from CSU-node signature ✓

frames 4

REPLY — PING_RESPONSE

Prefilled from the request: origin and target swapped, record_udt now, record_pagination + 1, parsed seeded per record key, references[0] holding the request head.

ping_response * tcp ping_response schema

tcp * url to tcp version infos pattern, max 120
https://tcp.digital/schema/1.3.0/

head * unified head informations

record_type * tcp record pattern, max 50, min 3
ping

record_direction * request|response|internal pattern, max 8, min 3
response

record_udt * universal datetime
09/06/2026 01:35:45 PM

record_pagination * unique creating number ≥ 1
45331

record_version * version pattern, max 10, min 5
1.3.0

origin_type * tcp type of origin pattern, max 3, min 3
TMH

origin_id * tcp id of origin pattern, max 120, min 3
P:ECS-HOST

target_type * tcp type of target pattern, max 3, min 3
CSU

target_id * tcp id of target pattern, max 120, min 3
E:DE*PID+E*IWS*00100017

context_type * tcp type of context / can be empty pattern, max 3
INF

context_id * id of context / can be empty pattern, max 120
6:90676293-fa28-49bd-81ba-1b58a0f62abd

error_code * tcp error handling. Hex pattern
000000

Load... Save... Reset reply

FIGURE 13 · The reply block. Note the swapped `origin_type` / `target_type` and the raised `record_pagination`. `Load...` and `Save...` here address the *response* schema.

The address to answer on is pre-filled and editable. If the sender advertised a return address you are told so; if the node had to infer it from the connection, you are told to check it. `Sign & Send response` signs and delivers it, and `Reset reply` seeds the form from the request again.

A reply you have started is kept per frame while the tab is open, so moving to another frame and back does not lose it. When the opened frame is itself a *response*, there is no form, just a note pointing at the request it answers.

Sent

Sent is the same layout for outgoing frames. There are no unread marks; what matters here is whether delivery worked.

The screenshot shows the TCPP Node interface with the 'Sent' tab selected. The left sidebar contains 'Compose', 'Inbox', 'Sent', 'Peers', 'Identity', and 'Log'. The 'Sent' tab shows a list of outgoing frames. The top section, 'Sent to any 2 peers', lists two peers: 'TMH-node localhost:8444' and 'localhost:9099 unknown node'. The 'Frames' section shows four outgoing frames:

- ping REQUEST** to TMH-node: signed by us, 202, 13:35:41, answered.
- charge_event REQUEST** to localhost:9099: signed by us, send failed, 13:35:23, awaiting response.
- measurement_create REQUEST** to TMH-node: signed by us, 202, 13:35:23, awaiting response.
- charge_start REQUEST** to TMH-node: signed by us, 202, 13:35:23, awaiting response.

The right pane shows the 'ping_request' details for the selected frame, including the payload, JWT, header, and transport tabs. The payload is a JSON object:

```
{
  "tcp": "https://tcp.digital/schema/1.3.0/",
  "head": {
    "record_type": "ping",
    "record_direction": "request",
    "record_uat": "2018-07-27T13:30:07.000Z",
    "record_pagination": 48330,
    "record_version": "1.3.0",
    "origin_type": "CSU",
    "origin_id": "E:DE*PID+E:IWS*00100017",
    "target_type": "TMH",
    "target_id": "P:ECS-HOST",
    "context_type": "INF",
    "context_id": "G:90676293-fa28-49bd-81ba-1b58a0f62abd",
    "error_code": "000000",
    "error_text": null
  },
  "record": {
    "ping": "Check alive"
  },
  "participants": [],
  "references": []
}
```

FIGURE 14 · Outgoing frames. signed by us on all of them, 202 where the peer accepted, and send failed where nothing was listening.

A frame is recorded whether or not it was delivered, so a failed send leaves a full record, token included, and the reason sits in the **Transport** tab. Once the answer arrives, the request picks up its answered line.

Peers and verification

A peer is a node you have exchanged frames with, or one you added yourself: a name, a `host:port`, and optionally its public key. Peers appear on their own the first time a frame passes in either direction, but **never with a key**. Supplying the key is always a deliberate act.

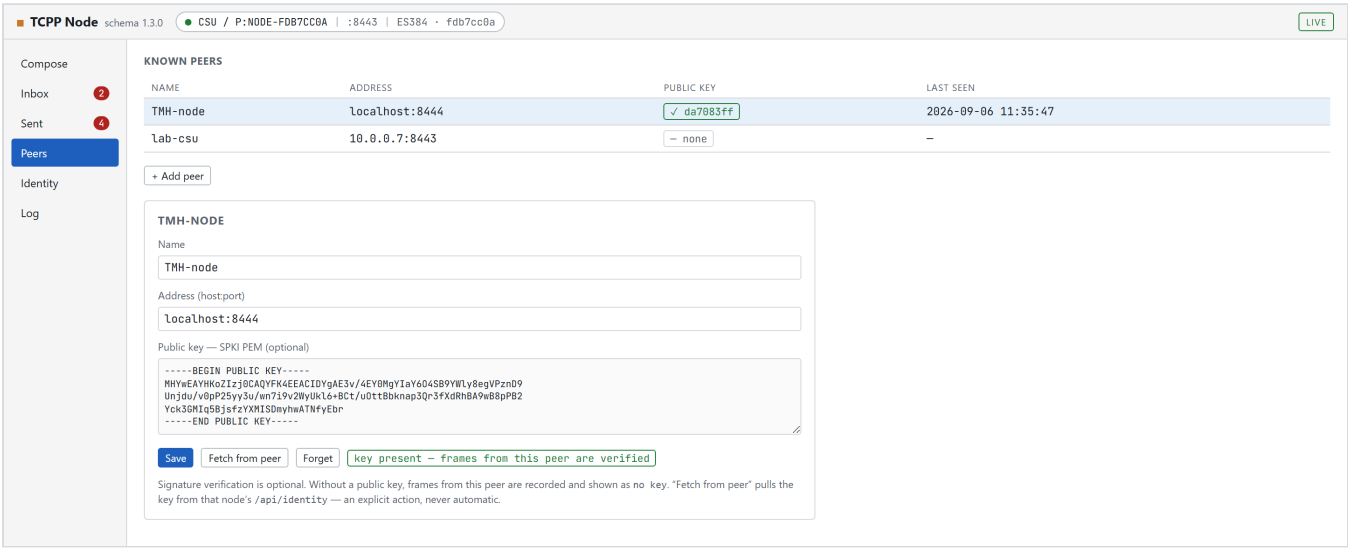


FIGURE 15 • Two peers: one whose key has been fetched, one that has never been contacted. Selecting a row opens it in the card below.

Giving a peer its key

- 01 Select the peer in the table, or press `+ Add peer` and fill in a name and address.
- 02 Either paste its SPKI PEM into the key box, or press `Fetch from peer` to pull it from that node's `/api/identity`. The peer must be saved before it can be fetched from.
- 03 Press `Save`. Every stored frame from that peer is re-checked immediately, and the badge under the box turns green.

The PEM to paste is on the other node's `Identity` screen, behind `Copy PEM`. `Forget` removes the peer; the frames it sent stay in the log and fall back to `no key`.

What the badges mean

BADGE	MEANING
signature ✓	The token verifies against the key stored for this peer.

BADGE	MEANING
signature ✕	It does not. The frame was still recorded; look at it.
no key	No key is configured for this peer, so nothing was checked.
unverified	Not checked, for example a frame whose token could not be parsed.
signed by us	This node signed it. Shown on everything under Sent .

VERIFICATION IS OPTIONAL; SIGNING IS NOT

A frame from a peer with no key is recorded and displayed, not rejected; you can run a whole lab without configuring a single key. Signing, on the other hand, always happens: every frame this node sends is an ES384 JWS.

Identity

The screenshot shows the 'Identity' tab of the TCPP Node interface. The top bar indicates the node is 'LIVE' and shows the schema version '1.3.0'. The left sidebar has a navigation menu with 'Compose', 'Inbox', 'Sent', 'Peers', 'Identity' (selected), and 'Log'. The main content area is titled 'THIS NODE' and contains three sections:

- Node Details:** A table listing node information.

Node	CSU / P:NODE-FDB7CC0A
Advertised	localhost:8443 - sent as X-TCPP-From so peers can reply
Algorithm	ES384 (ECDSA P-384 / SHA-384)
Key id	fdb7cc0accbb44b8
Fingerprint	47:3b:43:39:3a:91:23:15:25:fc:c9:d7:31:3a:a7:12:67:d4:74:27:2f:57:8e:78:52:0b:c3:4b:43:68:fa:5e
Certificate	self-signed, valid to 2036-09-03
Started	2026-09-06 11:35:17
- PUBLIC KEY — GIVE THIS TO PEERS THAT WANT TO VERIFY YOU:** A text box containing a PEM-formatted public key, with a 'Copy PEM' button below it.
- FILES NEXT TO THE BINARY:** A table listing the file paths for various node components.

Private key	C:\Users\User\tcpp-lab\tcpp-key.pem
Certificate	C:\Users\User\tcpp-lab\tcpp-cert.pem
Node name	C:\Users\User\tcpp-lab\tcpp-node.json
Frame log	C:\Users\User\tcpp-lab\tcpp-frames.jsonl
Peers	C:\Users\User\tcpp-lab\tcpp-peers.json
Records	C:\Users\User\tcpp-lab\tcpp-records

Below the file paths table, a note states: 'Drop your own tcpp-key.pem and tcpp-cert.pem next to the binary and restart to use them — the same pair signs JWTs and serves TLS.'

FIGURE 16 • Everything this node is, and where it keeps it.

Node

The TCPP module type and id this node signs and speaks as.

Advertised

The address sent to peers as `X-TCPP-From` so they know where to reply. Change it with `--advertise` when peers reach you by another name.

Algorithm

Always ES384: ECDSA on P-384 with SHA-384.

Key id

Derived from the public key. Its first eight characters appear in every token's JOSE header, which is how you tell at a glance which key signed a frame.

Fingerprint

SHA-256 over the certificate, for comparing out of band.

Certificate

Self-signed, ten years, generated on first run.

The middle card holds the public key to hand to anyone who wants to verify this node; [Copy PEM](#) puts it on the clipboard. The bottom card lists the six paths from [Getting started](#), resolved.

BRINGING YOUR OWN KEY

The same pair signs JWTs and terminates TLS, so using your own identity is just dropping `tcpp-key.pem` and `tcpp-cert.pem` next to the binary and restarting. A key with no certificate gets a fresh certificate derived from it. A certificate that does not match the key is a startup error rather than a silent inconsistency, and a certificate without its key is refused outright, because a certificate cannot sign anything.

Log

The screenshot shows the TCP Node Log interface. On the left is a sidebar with 'Compose' and 'Log' buttons. The main area is titled 'FRAME LOG' and contains a table with 6 frames. The table has columns for timestamp, direction (SENT/RECV), peer address, record type, and state. Below the table is a note about the log file location and format.

Timestamp	Direction	Peer Address	Record Type	State
2026-09-06 11:35:47	RECV	localhost:8444	ping response	signature ✓
2026-09-06 11:35:41	SENT	localhost:8444	ping request	202
2026-09-06 11:35:23	SENT	localhost:9099	change_event request	ERR
2026-09-06 11:35:23	RECV	localhost:8444	status request	signature ✓
2026-09-06 11:35:23	SENT	localhost:8444	measurement_create request	202
2026-09-06 11:35:23	SENT	localhost:8444	change_start request	202

appended to C:\Users\User\tcpp-lab\atcpp-frames.jsonl - one JSON object per line, tailed here

FIGURE 17 • Both directions in one place. The line at the bottom names the file these lines are tailed from.

Each line is a timestamp, **SENT** or **RECV**, the peer address, the record type and direction, and a state column: the HTTP status or **ERR** for outgoing frames, the verification result for incoming ones. Filter by direction or peer, search across record type, origin, target and frame id, and press **Refresh** to pull the list again.

The file behind it

`tcpp-frames.jsonl` is append-only: one JSON object per line, never rewritten, recording the verification state *as it was at that moment*. On start the node replays the last `--log-tail` lines (500 by default) so correlation and history survive a restart, re-checking signatures against the keys you hold now. Nothing is unread after a restart. Lines it cannot parse are skipped with a warning rather than stopping the node.

■ Workflows

A. Send your first request

- 01 Open **Compose** and select `ping` in the schema list.
- 02 Press **Load...** and pick `example`. The form fills and the badge turns `valid`.
- 03 Edit anything you like; `origin_id` and `target_id` are the usual first stops.
- 04 Check **Send to**. It holds the default target; the dropdown offers known peers.
- 05 Press **Sign & Send**. `✓ 202 accepted` means the peer took it.
- 06 Open **Sent** to see the frame, its token and its transport result.

B. Answer a request that arrived

- 01 On the receiving node, open **Inbox**. Unread frames carry a filled dot and the rail badge counts them.
- 02 Select the frame. It is marked read and opens on the right.
- 03 Read it on **Payload**; check who signed it with the badge in the header line.
- 04 Scroll down to **Reply**. The response is pre-filled from the request; fill in the parts that carry your answer.
- 05 Confirm the return address, then press **Sign & Send response**.
- 06 Back on the first node the response arrives in its inbox, and its request now reads `↳ answered`.

C. Turn `no key` into `signature ✓`

- 01 Open **Peers** and select the peer, or add it with its `host:port`.
- 02 Give it a name you will recognise in the frame lists, and press **Save**.

03 Press **Fetch from peer** (or paste the PEM from that node's **Identity** screen) and **Save** again.

04 Go back to **Inbox**: every frame that peer ever sent now reads **signature ✓**. Verification is recomputed over the whole stored log, not just new arrivals.

D. Keep a payload you will need again

01 Compose the request until it says what you want.

02 Press **Save...**, give it a name (**lab ping**, **ac-start-berlin**), press **Save**.

03 It appears under **Load...** marked **saved**, and on disk at `records/<schema>/<name>.json`.

04 Records are plain files. Copy the directory to another node and its library comes with it.

E. Test what a peer does with a bad message

01 Load a record, then break one field on purpose: an id that fails its pattern, a required field emptied.

02 The field turns red and the badge counts the issue. Send it anyway.

03 On the receiving node the frame is recorded like any other; what its operator sees is exactly what your peer implementation would see.

04 An unparseable token is recorded too, marked **unverified**, and answered with `400` instead of `202`.

F. Move a node, or give it a new identity

01 To move a node, stop it and copy its whole directory. Key, certificate, name, log, peers and records all travel together, so it comes back as the same node.

- 02 To hand it a new identity, drop your `tcpp-key.pem` and `tcpp-cert.pem` in and restart.
- 03 To have a fresh one generated, delete both files and restart; the node id is derived from the new key.
- 04 Check what it became with `./tcpp-node --print-identity`, then give the printed PEM to the peers that verify you.

THE KEY IS THE NODE

`tcpp-key.pem` is a secret and it is not recoverable. A new key means a new key id and a new node id, and every peer that verifies you has to be given the new public key.

■ Command line and files

Flags

There are no subcommands and no environment variables; flags and the files on disk are the whole configuration. Both `-flag` and `--flag` work.

FLAG	DEFAULT	MEANING
<code>--port</code>	<code>8443</code>	TLS port to listen on.
<code>--data-dir</code>	beside the binary	Where the key, certificate, log, peers and records live.
<code>--advertise</code>	<code>localhost:<port></code>	Address peers should reply to, sent as <code>X-TCPP-From</code> .
<code>--peer</code>	<code>localhost:<port></code>	Default target pre-filled in the browser. The default is this node itself.
<code>--node-type</code>	<code>CSU</code> , or the stored value	Three uppercase letters. Persisted after the first run.
<code>--node-id</code>	derived from the key	3–120 characters. Persisted after the first run.
<code>--hosts</code>	none	Extra certificate host names or IPs, comma separated.
<code>--log-tail</code>	<code>500</code>	Frames replayed from the log at startup.
<code>--insecure-peers</code>	<code>true</code>	Accept peers' self-signed certificates. Turn off with <code>--insecure-peers=false</code> .
<code>--print-identity</code>	<code>false</code>	Create or load the identity, print it and exit without listening.

The certificate a node generates always covers `localhost`, `127.0.0.1`, `::1` and the machine's host name; anything else peers will use to reach it belongs in `--hosts`.

KEEP ONE NODE PER DIRECTORY

Two copies of the binary started in the *same* directory would share one key, one frame log and one record library, and the second would fail to take the port. Give each node its own directory, or its own `--data-dir`.

--print-identity

```
$ ./tcpp-node --print-identity
node      CSU / P:NODE-74734B2A
alg       ES384
kid       74734b2a3dc9c8f9
fingerprint 48:81:2b:67:07:83:91:d7:ce:59:20:9a:83:0c:07:fa:ac:0f:da:f0:6a:da:02:6c
key       C:\Users\User\tcpp-lab\a\tcpp-key.pem
cert      C:\Users\User\tcpp-lab\a\tcpp-cert.pem (valid to 2036-08-31T16:33:05Z)

-----BEGIN PUBLIC KEY-----
MHYwEAYHKoZIzj0CAQYFK4EEACIDYgAEnfz9/7TBzZv7NCAdmkGv+KzYUVnfVeum
amuA/gGEdCStd2Tm7X1JJALLWPYN7dJ9LzNIX4qNPUIMiRy0KZkif28ZvL6iuRTr
+voCuitlwh+WBuGNBFiCSvFQcFbfjDpy
-----END PUBLIC KEY-----
```

It creates the identity if there is none, which makes it a convenient way to provision a node before its first start.

Running and stopping

The node runs in the foreground and logs to standard error. `Ctrl+C` (or `SIGTERM`) shuts it down gracefully within five seconds. It exits `0` on a clean shutdown and `1` after printing `tcppnode: <reason>`, be it a port already in use, a data directory it cannot write, or a certificate that does not match its key.

What to back up

`tcpp-key.pem` and `tcpp-cert.pem` are the identity and cannot be regenerated. `tcpp-node.json`, `tcpp-peers.json` and `records/` are configuration worth keeping. `tcpp-frames.jsonl` is history: it grows forever and can be rotated when it gets inconvenient; the node only ever reads its tail.

■ HTTP API

The browser application is an ordinary client of the node's own HTTP interface, so anything it does can be scripted. All of it is served over TLS on the node's single port.

NO AUTHENTICATION

None of these routes require credentials, and there are no origin checks. Treat reachability as full control of the node.

Node to node

ROUTE	MEANING
<code>POST /tcpp</code>	The body is a compact JWS, <code>Content-Type: application/jwt</code> , up to 1 MB. Send <code>X-TCPP-From</code> so the receiver can reply and <code>X-TCPP-Node</code> to name yourself. Answers <code>202</code> with <code>{"received":"<frame id>","node":"<label>"}</code> , or <code>400</code> if the token cannot be read, in which case the frame is still recorded.

Browser

ROUTE	RETURNS
<code>GET /</code>	The web application, embedded in the binary.
<code>GET /schema/1.3.0/{name}</code>	One embedded schema, as <code>application/schema+json</code> . <code>.json</code> suffix is accepted.
<code>GET /api/identity</code>	Node type and id, label, advertised address, default peer, key PEM, <code>kid</code> , fingerprint, certificate expiry, file paths, state frame and unread counts.
<code>GET /api/catalog</code>	Every endpoint with its request and response schema names and example variants.
<code>GET /api/example/{base}[/{variant}]</code>	One shipped example payload. Omitting the variant means <code>example</code> .
<code>POST /api/send</code>	Signs and delivers. Body <code>{"peer":"host:port","message":{...},"in_reply_to":"<frame id>"}</code> . Always <code>200</code> , with

ROUTE	RETURNS
	recorded <code>frame</code> , the <code>jwt</code> and a <code>transport</code> object; <code>err</code> present when delivery failed.
<code>GET /api/frames</code>	Frame list plus per-peer counts and totals. Filters: <code>?direction=sent received</code> , <code>?peer=host:port</code> , <code>?limit:</code>
<code>GET /api/frames/{id}</code>	One frame in full: payload, token, decoded JOSE header, a reply schema when there is one.
<code>POST /api/frames/{id}/read</code> <code>POST /api/frames/read</code>	Mark one frame, or all of them, read.
<code>POST /api/frames/{id}/verify</code>	Re-check a signature. An optional body <code>{"public_key_pem": "..."} </code> checks against a key you supply instead of the stored one.
<code>GET /api/peers</code>	Known peers with name, address, key state, <code>kid</code> , node id seen.
<code>PUT /api/peers</code>	Create or update: <code>{"name", "address", "public_key_pem", "node_id", "re</code> Saving re-verifies every stored frame from that peer.
<code>DELETE /api/peers/{name}</code>	Forget a peer.
<code>POST /api/peers/{name}/fetch-key</code>	Read the peer's <code>/api/identity</code> and store its public key.
<code>GET /api/records/{schema}</code>	The records stored for one schema, each with <code>shipped</code> , <code>modified</code> , <code>size</code> .
<code>GET /api/records/{schema}/{name}</code>	One record, byte for byte as stored.
<code>PUT /api/records/{schema}/{name}</code>	Store one record. The body is the raw JSON payload, up to
<code>DELETE /api/records/{schema}/{name}</code>	Remove one record.
<code>GET /api/events</code>	Server-sent events: <code>frame.received</code> , <code>frame.sent</code> , <code>frame.updated</code> .

Driving a node from a shell

The node's certificate is self-signed, so a client has to be told to accept it.

```
# the node's own identity
curl -k https://localhost:8443/api/identity

# send a stored record to a peer
curl -k -X POST https://localhost:8443/api/send \
  -H 'content-type: application/json' \
  -d "{\"peer\":\"localhost:8444\",\"message\":$(curl -sk https://localhost:8443/ap

# what has arrived
curl -k 'https://localhost:8443/api/frames?direction=received&limit=10'
```

```
# PowerShell
$node = 'https://localhost:8443'
$msg = Invoke-RestMethod "$node/api/records/ping_request/example" -SkipCertificate
Invoke-RestMethod "$node/api/send" -Method Post -SkipCertificateCheck `
  -ContentType 'application/json' `
  -Body (@{ peer = 'localhost:8444'; message = $msg } | ConvertTo-Json -Depth 20)
```

PAYLOAD BYTES ARE NEVER RE-ENCODED

`/api/send` signs the `message` exactly as you hand it over, key order included, and a receiver reads it back verbatim. That is deliberate: a signature covers bytes, not an abstract object, so nothing is allowed to re-serialise a message between signing and verification.

■ The TCPP message

TCPP is the protocol these schemas define: a JSON envelope exchanged between charge infrastructure modules. This manual's version is **1.3.0**, and every message names it in its first field.

```
{
  "tcpp": "https://tcpp.digital/schema/1.3.0/",
  "head": {
    "record_type": "ping",
    "record_direction": "request", request | response | internal
    "record_udt": "2018-07-27T13:30:07.000Z",
    "record_pagination": 45330, counts up, 1 to endless
    "record_version": "1.3.0",
    "origin_type": "CSU", three letters, the sending module
    "origin_id": "E:DE*PID*E*IWS*00100017",
    "target_type": "TMH",
    "target_id": "P:ECS-HOST",
    "context_type": "INF", the process, not the module
    "context_id": "G:90676293-fa28-49bd-81ba-1b58a0f62abd",
    "error_code": "000000" hex; 000000 = no error
  },
  "record": { "ping": "Check alive" }, shape depends on record_type
  "parsed": { }, responses only: an error code per elem
  "participants": [ ], controls outgoing communication
  "references": [ ] a response carries the request's head
}
```

Ids carry their namespace

Every id starts with a two-character prefix saying what kind of identifier follows.

PREFIX	MEANING	PREFIX	MEANING
E:	EVSEID / EMAID	G:	GUID
D:	domain namespace	H:	hash
P:	private namespace	N:	numeric pagination
I:	internal	C:	contract / tariff id

PREFIX	MEANING	PREFIX	MEANING
V:	VID	X:	undefined

A node that generates its own id uses the private namespace, which is why the lab nodes are called `P:NODE-9BED6853`.

Module types

`origin_type` and `target_type` are three uppercase letters naming the kind of module: `TMH` Trusted Management Host, `TEH` Trusted Exchange Host, `TAH` Trusted Archive Host, `MSP` eMobility Service Provider, `CP0` Charge Point Operator, `CSU` Charge Switch Unit, `EMU` Energy Management Unit, and so on. A node announces its own with `--node-type`.

How a request and its answer find each other

Every request has exactly one response. The responder copies the request's whole `head` into `references[0]`, and both nodes compute the same key from it: record type, direction, timestamp, pagination, origin and target. That is how the frame lists on both sides can say `↳ answered` and `↳ answers an earlier request` without either node keeping a session. When you reply from the inbox the node also passes the frame id explicitly, so the pairing does not depend on the timestamps surviving intact.

Signing

An outgoing frame is a compact JWS over the payload bytes, with the header `{"alg":"ES384","kid":"...","typ":"JWT"}`. Two details are worth knowing because they are where interoperable implementations usually go wrong:

- **Fixed-width R||S.** JWS wants the ECDSA signature as two 48-byte halves (IEEE P1363), which is what the browser's Web Crypto produces. Go's `ecdsa.SignASN1` produces DER instead and must not be used; roughly one signature in 256 has a short coordinate that needs left-padding.
- **The payload is signed as handed over.** `JSON.stringify` preserves key insertion order while Go's `encoding/json` sorts map keys, so a message is never re-marshalled between signing and verification.

■ Appendix

The 27 endpoints

Each has a request and a response schema, served at

`/schema/1.3.0/<base>_request.json` and `..._response.json`. Some ship more than one example, and each example becomes a record you can load.

ENDPOINT	EXTRA EXAMPLE VARIANTS
<code>archive_get</code> , <code>archive_list</code>	none
<code>charge_event</code>	request: <code>example_with_charge_end</code>
<code>charge_process</code> , <code>charge_start</code>	none
<code>display_input</code> , <code>display_output</code>	none
<code>emaid_get</code>	request: <code>example_via_vehicle</code>
<code>error</code>	request: <code>example_hardware_internal</code> , <code>example_hardware_request</code>
<code>key_get</code>	none
<code>measurement_change</code> , <code>measurement_configuration</code> , <code>measurement_create</code>	none
<code>module_get</code>	none
<code>parameters_delete</code> , <code>parameters_get</code> , <code>parameters_set</code>	none
<code>parameters_list</code>	response: <code>example_measurement</code>
<code>ping</code>	response: <code>example_with_participants</code> , <code>example_with_reference_jws</code>
<code>status</code>	none
<code>tariffs_list</code>	request: <code>example_cp_in_location</code> , <code>example_cp_search</code>

ENDPOINT	EXTRA EXAMPLE VARIANTS
<code>trigger_delete</code> , <code>trigger_get</code> , <code>trigger_set</code> , <code>triggers_list</code>	none
<code>verification_get</code> , <code>verification_set</code>	none

Glossary

Frame

One TCP message as this node saw it: token, decoded payload, direction, peer, verification state, transport result and correlation links.

Peer

Another node: a name, an address, and optionally a public key.

Record

A stored payload in the node's library, at `records/<schema>/<name>.json`.

Draft

Unsent form contents kept in the browser tab. Per schema on the compose form, per frame on the reply form.

Target

The `host:port` a frame is sent to.

kid

Key id, derived from the public key and carried in every token's header.

JWS / JWT

The compact signed form a frame travels in. Here always ES384.

About the figures

Every screenshot in this manual was taken from a running two-node lab, not drawn, so what you see here is what the application does. They are captured by a script kept alongside the manual's source, which drives a headless browser through the same steps an operator takes, and they are retaken whenever the interface changes.

tcpp.digital English user manual · schema 1.3.0

Figures captured from a two-node lab (CSU on 8443, TMH on 8444) on 3 September 2026. Parts of these documents have been authored with AI assistance.