qchat — command line manual

For the person who installs, runs and drives a qchat node from a terminal. It covers the three programs this repository produces and nothing else.

Applies to	`qchat` 1.0.0
Programs	`qchat` (the node), `qchatctl` (its client), `qchatsign` (the release key)
Audience	Whoever has a shell on the machine that runs the node
Also as	PDF, attached to every release of this repository

> **Pending.** This manual describes the command line only. The web console, `qchatui`, is a separate repository and its own screens are not documented here yet. **This manual must be extended with them** once that console is released: what each screen does, what it shows that the terminal does not, and how a person is expected to move between the two. Until then, the section The console says only how to install it and where to reach it.

Contents

What qchat is

qchat is a node of the Cuezero network that a person runs on their own machine. Its conversations are chains: every message is a transaction sealed under a key only the members of that moment hold, every block is certified by the members who write, and the history a node shows is what its copy of the chain says. There is no server in the middle; the seeds of the network only introduce nodes to each other.

Around that, qchat adds what a chat needs and a chain does not have: a name and a status, contacts, the difference between a message that left and one that was read, somebody typing, files shared into a conversation, files kept in a vault only the owner opens, and an explorer that reads the chains on the machine.

Before you start

Requirement	Why
Ports 7420/udp, 7421/udp, 7422/tcp open, or a router that punches	Without them the node reaches nobody. A home router usually does; a corporate one may not.
A clock that is roughly right	Every certificate, profile and token assumes the machines agree on the time within minutes.
Minutes of CPU, once	Solving the identity is deliberately expensive at the cost of the production network. It happens once.
A terminal on the same machine, or an SSH tunnel	The control surface and the console are on the loopback.

Installing

From a release, on Debian and its derivatives:

```
sudo dpkg -i qchat_1.0.0_amd64.deb
```

That installs `qchat`, `qchatctl` and `qchatsign` in `/usr/local/bin`, the sample configuration in `/usr/share/doc/qchat/`, the unit in `/usr/lib/systemd/system/` and the console deployment script in `/usr/share/qchat/`.

From a release, as an AppImage: download it, make it executable, run it. It carries the node and the client and swaps itself with a verified update.

From source:

```
git clone https://xato.youser.app/xatoxi-capps/qchat.git
cd qchat
./setup_repo.sh --prefix "$PWD/.local/xato" --clean-prefix --clean-deps-build
```

That leaves the three programs in `build/`.

First start, step by step

1. The identity

```
qchat --show-id
```

The first time, this solves the identity, which takes minutes, and writes it under `~/.local/share/qchat/`. Then it prints two lines:

```
node   3f9a…   what other nodes connect to; what a contact adds
member 8c21…   what conversations list; what an invitation names
```

Every later call prints the same two lines at once. **Give the node id to the people who should reach you, and the member id to whoever founds a conversation with you.**

2. The node

```
qchat
```

With no file, it joins the production network, serves its console on `http://127.0.0.1:7423/` and its control surface on `ws://127.0.0.1:7424/ws`, and writes the token of that surface to `~/.local/share/qchat/ws.token` with mode 0600. It runs until `Ctrl-C`.

3. A profile and a contact, from another terminal

```
qchatctl profile-set "Alice" at the laboratory
qchatctl friend-invite <node id of Bob> hi, this is Alice
```

On Bob's side:

```
qchatctl friend-requests
qchatctl friend-accept <id the previous one printed>
qchatctl friend-list
```

The last one shows Alice with `ready` once the link is up and `hasProfile` once her record arrived. Only one of the two types an identifier: the other answers with the short id of the request.

4. A conversation

```
qchatctl room-create direct "" <member id of Bob>
qchatctl msg-send <convId> hola Bob
qchatctl msg-history <convId>
```

The configuration file

Default location `/etc/qchat/qchat.cfg`; `--config` names another, and a node run by a person usually keeps its file beside its data and names it: `qchat --config ~/.local/share/qchat/qchat.cfg`. Every setting has a default and a file with none of them is a valid file. The sample in

`deploy/qchat.cfg.sample` lists every setting with its default and a comment; this is the short form.

Group `qchat`	**Default**	**What**
`dataDir`	`~/.local/share/qchat`	Root of every file below
`identityPath`, `keyringPath`	`identity.cnis`, `keyring.bin`	The identity and the reading key pair
`storePath`, `ledgerPath`	`store`, `ledger`	The local store and the chains
`tokenPath`	`ws.token`	The token of the control surface
`runtimeDir`, `uiDocRoot`, `downloadDir`	`run`, `ui`, `downloads`	Derived files, the console, what is fetched
`storeMapMb`, `ledgerMapMb`	256, 4096	Ceilings of the two maps
`network`	`mainnet`	`mainnet` (`prod` is taken as its old spelling), `dev`, `custom` or `standalone`
`bindAddr`, `publicAddr`	`0.0.0.0`, none	Where the node binds; what it publishes behind a 1:1 NAT
`dhtPort`, `relayPort`, `ipv6`	7420, 7422, 1	The peer to peer ports; 7421 is fixed one above the first
`maxLinks`	128	Links kept open at once
`puzzleStatic`, `puzzleDynamic`	22, 18	The cost of an identity; a property of the network
`httpPort`, `wsPort`, `wsAddr`	7423, 7424, `127.0.0.1`	The console, the surface, where they bind
`wsQblock`	1	Serve the explorer on the surface
`authGraceMs`, `pushMs`, `maxAdminConns`	5000, 1000, 16	Time to present the token, push period, clients at once
`roundMs`	4000	A round of the agreement, after which the next proposer takes the turn
`fileMaxMb`, `vaultFragLen`	16, 4096	Largest shared file; vault fragment size
`typingSend`, `typingShow`, `readReceipts`	1, 1, 1	Three of the switches at start
`profilePublic`	0	The fourth: who sees your name, contacts (0) or everybody (1)
`cardMs`	600000	How often a member’s public card is looked up; yours goes out every two

Group `qchat`	Default	What
`logLevel`, `logEnabled`	`INFO`, 1	`TRACE`, `DEBUG`, `INFO`, `WARNING` or `ERROR`
`uiVersion`, `uiSha256`, `uiRegistryUrl`	none	Only to deploy a console other than the one the release carries; the version keeps its `v`
`updateUrl`	the project registry	Where the signed manifest is read

Group `bootstrap`	What
`seed0` … `seed7`	`host:port/pin[@asn]`, read only when `network` is `custom`

A file that is refused names the code: `QCHAT_ERR_CFG_PORT`, `QCHAT_ERR_CFG_NETWORK`, `QCHAT_ERR_CFG_LOG`, `QCHAT_ERR_CFG_CENSUS` and so on, and the node does not start.

The node: `qchat`

```
qchat [options]
  --config <path>    Configuration file
  --network <name>   Join mainnet, dev, custom or standalone
  --standalone       Start a network instead of joining one
  --show-id          Print the node and member identities and stop
  --quiet            Keep the log off; only answers on standard output
  --version          Print the version and stop
  --help             Print this and stop
```

`--network` on the command line wins over the file. `--standalone` is `--network standalone`. Both `SIGINT` and `SIGTERM` stop it cleanly: the chains are closed, the store is flushed, the ports are released. `SIGHUP` is ignored on purpose; a change of configuration is a restart.

What it does at start, in order: reads the file, opens the identity (solving it if absent), opens the store and the chains, builds the node, the agreement and the ledger, wires the files, loads the switches, the profiles and the conversations it followed, opens the ports, then the control surface, the console and the updater. Anything refused in that chain stops it with the name of the code on the standard error.

The ports

Port	Protocol	Who	Purpose
7420	UDP	the network	The routing table and the handshake
7421	UDP	the network	The data plane, fixed one above the first
7422	TCP	the network	Relay, when a direct path cannot be made
7423	TCP, loopback	you	The console, static files

Port	Protocol	Who	Purpose
7424	TCP, loopback	you	The control surface, WebSocket

The last two must **not** be reachable from outside. The console is a static bundle with no state of its own; the surface is where every action goes, and its only guard is the token. Administration from another machine is an SSH tunnel:

```
ssh -L 7424:127.0.0.1:7424 -L 7423:127.0.0.1:7423 user@machine
```

with a copy of `ws.token` on the near side, named in the client's configuration.

Driving it: `qchatctl`

```
qchatctl [options] <command> [words]
  --config <path>   Configuration file (the same one qchat reads)
  --json            Print the answer as JSON
  --follow          Stay subscribed and print every event
  --verbose         Log at the information level
  --quiet           Keep the log off
  --help            Print this and stop
```

The client reads the same configuration file to find the port and the token, opens the surface, presents the token, sends one action and prints the answer: one field per line, or the JSON payload with `--json`. A refusal prints `qchatctl: <code name> (<number>)` on the standard error and the exit status is 1.

`--follow` subscribes and prints every event as it comes, until the node closes the link or you press `Ctrl-C`. With `--json` every event is one JSON line, which is what a script wants.

Every command below is one action of the catalogue the surface serves. Words in brackets are optional; `yes|no` also accepts `1|0`, `true|false`, `on|off`.

Conversations

```
rooms
room-create <kind> [name] [member,member...] [role,role...]
room-join <convId> <creator> [token]
room-leave <convId>
room-rename <convId> <name...>
room-open <convId> [yes|no]
room-members <convId>
room-invite <convId> [role] [ttlSec] [maxUses]
member-add <convId> <member> [role]
member-remove <convId> <member>
member-role <convId> <member> <role>
```

Kinds. `direct` is two people and both write. `group` is a team where every `writer` writes and a `reader` reads. `bulletin` is a community where the founder alone writes. The kind is fixed at creation.

Roles. `owner` (the founder, one per conversation), `writer`, `reader`. Members who write certify the blocks; readers hold the chain and the key and write nothing.

Founding. `room-create` writes the genesis, adds every member named, sends each a notice, and rotates the key once every member's reading key is known. `rooms` shows `ready` when that is done; until then messages are refused with `QCHAT_ERR_LGR_KEY_PENDING`. The members named must be contacts of yours whose profile you hold, and you must be a contact of theirs, or their nodes drop the notice.

Entering. A member put in by the founder enters by the notice, with nothing to do. Anybody else enters with an invitation: `room-invite` prints a token (hexadecimal) with a role, a life in seconds and a number of uses, and `room-join <convId> <creator member id> <token>` on the other side writes the join into the chain. The founder then rotates the key for the new membership and hands the newcomer the key of the current epoch. What was written in earlier epochs stays unreadable to the newcomer, and `msg-history` says `no key` for those entries. A token handed before the first block of the chain has arrived is kept and offered once it is in, so the join may be given right after the founder is a contact.

`room-join` without a token follows the chain without joining, which is what a reader of a bulletin does before being added.

The name. The name of a conversation is kept in its chain, sealed under the key of the epoch in force: only members read it, and every member reads it whichever way they came in. The owner writes it once the conversation is ready and again after every rotation, so a newcomer sees it shortly after being handed its key. A restart does not write it again when the chain already holds it for that epoch. Only the owner changes it, with `room-rename <convId> <name>`: the new name goes into the chain on the next pass and every member takes it when that is applied.

Leaving. `room-leave` writes the leave when you are a member and stops following in any case. The founder cannot leave a conversation with members in it; remove them first, or leave the conversation to end.

On screen. `room-open <convId> yes` says the thread is in front of you: from then on your node sends read receipts for it, if the switch is on, and `no` when you look away.

Messages and their states

```
msg-send <convId> <text...>
msg-history <convId> [from] [max] [hidden]
msg-delete <convId> <height> <index>
msg-recover <convId> <height> <index>
typing <convId> [yes|no]
```

`msg-send` seals the text under the key of the current epoch and hands it to the ledger. It answers at once; the message is `sending` until the members who write certify its block, which on a quiet network is a second or two.

`msg-history` prints a page from block `from` (1 is the start, 0 the default) of at most `max` messages (50 by default, 200 at most), cut at a block boundary so that the next page never repeats a message; `next` says where the next page starts and `more` whether there is one.

`hidden yes` includes the messages you hid. Every entry carries its block and index (what `msg-delete` takes), its author with the name you know them by, the epoch, the kind, the text, and:

state	What it means
`sending`	In the pool; no certified block holds it yet
`sent`	In a block certified by the writers
`delivered`	Every other member has that block
`read`	Every other member showed it on a screen

why	When a message could not be opened
`no key`	It was sealed under an epoch you have no key for; a late joiner sees this on what came before
`not a member`	You were not a member of that epoch
`tampered`	The key you hold opens nothing here

`msg-delete` hides a message on this machine and nowhere else: the chain is what it is. `msg-recover` shows it again.

`typing <convId> yes` tells the members you are writing; the node repeats it every three seconds while it lasts and the others stop showing it five seconds after the last one. `no` stops it early.

People: contacts, profiles, presence

```
friend-invite <node> [message...]
friend-remove <node>
friend-requests
friend-accept <id>
friend-reject <id>
friend-add <node>
friend-list
peer-status <node>
profile-get [member]
profile-set <name> [status...]
presence-set [typingSend] [typingShow] [readReceipts] [profilePublic]
net-status
```

A **contact** is made with a **request**. `friend-invite <node> [message]` asks that node to be your contact: your node opens a link and, once it is usable, sends the request with your signed profile and the message (up to 280 bytes). On the other side, `friend-requests` lists it with a short `id` (the first eight digits of the asker's node id), their name, status and the message; `friend-accept <id>` accepts it and answers with the own profile; `friend-reject <id>` forgets it and tells nobody. Until the yes arrives, the request is listed on the sender's side too, with `incoming no`, and goes out again every time the link recovers. A request is accepted only when it carries a profile that verifies against the sender: nobody asks in somebody else's name. When both ask each other, the second ask counts as a yes. Asking somebody who is a contact already sends nothing and says so: `QCHAT_ERR_PRF_FRIEND`. `friend-remove <node>` forgets a contact, a request or an acquaintance on this side alone and tells nobody: the other side keeps

this node until it forgets it too, and its next greeting is listed here as a request from a stranger. It is how an invitation is tried again from scratch.

`friend-add <node>` is the old way and is still there: it makes that node a contact on this side only and greets it with the own profile. Whoever gets that greeting from a stranger sees it in `friend-requests` as a request with no message.

Your node connects to every contact and asks for its profile every five seconds until it has one. `friend-list` shows each with `ready` (the link is usable), `hasProfile` and `friend` (a contact, not just an acquaintance), and the name and status once known.

A **profile** is a signed record: name, status line, an avatar of up to 8 KiB, the reading key of that node, and a sequence. `profile-set` changes yours; every contact learns it at their next request. `profile-get` with no member prints your own; with a member id prints theirs, if you hold it. A record whose signature does not verify, or that is older than the one held, is dropped.

Presence is four switches:

Switch	Off means
`typingSend`	Your node never says you are typing
`typingShow`	Your node ignores what others say about typing
`readReceipts`	Your node sends no read watermark, and therefore receives none: whoever sends none is sent none
`profilePublic`	Only your contacts see your name and status line; the members of your conversations see your member id. Off by default

`presence-set` with no words prints them; with `yes|no` words sets them in that order, and the change is kept across restarts.

Your profile, avatar included, goes to your contacts and to nobody else: a member you met only in a conversation who asks for it is not told. With `profilePublic yes` your node publishes a **card** of the profile in the network: name and status line, signed, without the avatar. Every node that shares a conversation with you looks it up and shows your name. The card goes out again every twenty minutes and expires an hour after the last time. With `profilePublic no` your node publishes a blank card in its place, and whoever reads it forgets your name; a node that missed it forgets the name when the public card expires. A card cannot be deleted from the nodes that keep it, only replaced.

`net-status` prints how the network sees this node: `inNetwork`, the links open and usable, the peers held, how the outside world can reach you (`reach`) and whether that was confirmed. `peer-status <node>` prints the path in use with one contact.

Files and the vault

```
file-share <convId> <path>
file-list [convId]
file-fetch <fileId>
vault-store <path> [name] [local|net]
vault-list
vault-fetch <name>
vault-export <name> <path>
vault-import <path>
```

A **shared file** is cut into chunks, sealed under the epoch key of the conversation, written into its chain as a series of transactions, and reassembled by every member with the key. `file-share` answers with the file id at once; the chunks follow in the next blocks. `file-list` shows each file with its name, size, chunks, how many this node has, and `complete`, `sending` or `failed`. `file-fetch` writes a complete file into the downloads directory and prints the path; a name already there gets a numeric suffix. The largest file is `fileMaxMb`.

A **vault file** is cut into fragments sealed under a key derived from your identity. `vault-store <path> [name]` keeps it under the name given or the file's own; `vault-list` shows the names, sizes, fragment counts and where they are (`where local` or `where network`); `vault-fetch <name>` writes it back into the downloads directory. An empty file is refused.

By default (`local`) the fragments **never leave this machine**. With `net` as the third word they are published as signed records into the record cache of the nodes of the network as well, and a sealed copy is kept here so they can be offered again: a custodian forgets a record within the hour unless it is re-offered, and the node does that every 20 minutes while it is up. The network work is done by the pump, one fragment per pass and one job at a time: `vault-store … net` answers as soon as the job is accepted, and `vault-list` shows how it goes (`state storing`, `done`/`total`) and how it ended (`state done` with `copies`, the fewest custodians a fragment found, or `state failed` with its `code`); the recipe is kept only when every fragment found a custodian. `vault-fetch` of a network file asks the network first and the local copy second: the first call starts the rebuild and answers `QCHAT_ERR_FIL_PENDING`, and the first call after it ended hands the path back; `served` says how many fragments the network served. The ceiling over the network is 256 KB (`QCHAT_ERR_FIL_NET_SIZE`), in fragments of 2048 bytes; another job running answers `QCHAT_ERR_FIL_NET_BUSY` and a node with no routing table yet `QCHAT_ERR_FIL_NET_DOWN`. Every step is told by the `vault-update` event.

`vault-export <name> <path>` writes the **sealed recipe** of a file and `vault-import <path>` takes one in. The recipe leaves as it is kept: only the identity that sealed it opens it, so it is safe to carry and useless to anybody else. It is what another machine of the same identity needs to rebuild a file whose fragments are in the network.

The explorer

```
chain-list
chain-info <convId>
block-list <convId> [from] [max] [desc]
block-show <convId> <height>
tx-list <convId> <height>
tx-show <convId> <height> [index]
chain-verify <convId> [height] [index]
```

The explorer reads the chains on this machine, decodes what it can and says what it cannot. It is served only when the surface binds the loopback and `wsQblock` is 1; otherwise every one of these answers `QCHAT_ERR_BLK_OFF`.

- `chain-list`: every chain held, with its name, kind, tip, blocks applied and bytes.
- `chain-info`: one chain in detail: founder, epoch, members, validators, tip and genesis hashes, partitions.
- `block-list`: a page of blocks from `from` (1 is the genesis), at most `max` (100 by default, 500 at most), ascending unless `desc`; each row with its hash, time, transactions and `hasCert`, whether a certificate is kept for it. A page checks no signature.
- `block-show`: one block whole: the row, its chaining and Merkle checks and its certificate judged as the ledger judges it: `writers`, who could write at the height below the block; `quorum`, how many of them had to sign; `certified`, whether that many signatures hold; and each signer with `verified`, whether its signature holds. None of it depends on who is a member today. The chain is replayed up to the block to answer, so the cost grows with the height.
- `tx-list` and `tx-show`: the transactions of a block and one of them decoded: a founding, a member added or removed, a key rotated, a message opened when you hold the key and marked `hidden` if you hid it.
- `chain-verify`: validates the chain from the genesis, audits every certificate, and with a height and an index proves that transaction against the Merkle root of its block.

The identity

```
identity-export <path> <passphrase>
identity-import <path> <passphrase>
```

`identity-export` writes the identity and the reading key pair into one file sealed under a passphrase it asks for (eight characters at least) and prints the path. `identity-import` opens that file on another installation and writes the identity beside that node’s; the files already there are kept with a `.bak` suffix. The running node keeps the identity it started with; the imported one is used from the next start. The wrong passphrase is `QCHAT_ERR_IDN_PASSPHRASE`; a file that is not an export is `QCHAT_ERR_IDN_FORMAT`.

Two machines running the same identity at once confuse everybody who has it as a contact. Import, then stop the old one.

Updates

```
update-check
update-apply [appimage|deb]
config-show
```

The node reads a signed manifest from `updateUrl` thirty seconds after starting and every six hours, verifies the signature against the key compiled into it, and pushes `update-available` to every subscriber when the version is newer. `update-check` does it now and prints the manifest with `verified` and `newer`. `update-apply` downloads the artifact of the kind asked (or the one that matches how the node runs), checks its digest and size, and: for an AppImage, replaces the file the node runs from and asks for a restart; for a `.deb`, leaves it in the downloads directory and prints the path for `dpkg -i`.

A build whose key is the placeholder refuses every manifest with `QCHAT_ERR_UPD_UNSIGNED`; a manifest whose signature does not verify is `QCHAT_ERR_UPD_SIG`; an artifact whose digest is wrong is `QCHAT_ERR_UPD_HASH` and is deleted. A verified manifest may carry a new census of seeds; it is written to `downloads/seeds.cfg` and read at the next start.

`config-show` prints the effective settings, without the token.

The console

The console is a static bundle served on `http://127.0.0.1:7423/`. It is not in this repository, but it **travels inside the release**: a machine that has just installed qchat serves the console that came with the program, with nothing configured and no network, and updating the program updates the console with it, because the two travel in the same signed artifact. There is nothing to do.

The node picks in this order and serves the first it finds:

1. whatever is deployed in `uiDocRoot`, because whoever put it there meant it;
2. the console the installation carries (`/usr/share/qchat/ui`, or inside the portable image);
3. the placeholder page, which says what is going on.

Deploying a console other than the one the release carries is the operator path, and that is what `uiVersion` and `uiSha256` are for:

```
./deploy/fetch-ui.sh --config ~/.local/share/qchat/qchat.cfg
```

The version is the **publication tag, with its** `v`: written without it the registry answers 404, which reads like an outage and is a typo. A digest that does not match deploys nothing. The console talks to the surface with the token; it does the same things `qchatctl` does, because there is one catalogue.

Everyday recipes

Is it alive?

```
qchatctl status
qchatctl net-status
```

Watch everything that happens:

```
qchatctl --follow --json subscribe
```

Found a team, invite somebody, see them come in:

```
qchatctl room-create group "team"
qchatctl room-invite <convId> writer 86400 5
# give the token to Carol; she runs:
qchatctl room-join <convId> <your member id> <token>
# you run:
qchatctl room-members <convId>
```

Read the newest twenty blocks of a chain:

```
qchatctl block-list <convId> 0 20 desc
```

Move to another machine:

```
qchatctl identity-export ./me.qid "the passphrase"
# copy the file; on the new machine:
qchatctl identity-import ./me.qid "the passphrase"
# stop the old node, start the new one
```

Run it as a service on a machine nobody sits at: see `deploy/qchat.service`, which explains the state directory, the user and the clock.

When something is wrong

Symptom	Look at
`qchatctl` says `QCHAT_ERR_CTL_CONNECT`	The node is not running, or `wsPort` in the client's file is not the node's
`qchatctl` says `QCHAT_ERR_CTL_DENIED`	The token file the client reads is not the node's
`net-status` says `inNetwork 0` for minutes	The peer to peer ports are blocked, or the census is wrong; the log names the seeds it tried
A contact never gets `ready`	They are offline, or neither of you can be reached and the relay is refused; `peer-status` says the path
A contact never gets `hasProfile`	Their node has not added you; a profile is answered to contacts
`rooms` never says `ready`	A member's reading key is unknown: they are not a contact with a profile
`msg-send` says `QCHAT_ERR_LGR_KEY_PENDING`	The rotation has not been applied yet; wait for `ready`

Symptom	Look at
A message stays `sending` for two minutes and vanishes	No quorum certified it: the writers are offline. It is dropped from the pool; send it again
`msg-history` says `no key`	That epoch was before you joined; nothing to fix
`chain-list` says `QCHAT_ERR_BLK_OFF`	The surface is not on the loopback, or `wsQblock` is 0
`update-check` says `QCHAT_ERR_UPD_UNSIGNED`	This build carries no release key; see `doc/release-signing.md`

The log is on the standard error of `qchat`; `logLevel = "DEBUG"` says why every refusal happened.

Error codes

Every answer that is refused carries the name and the number of one code.

Family	Range	Meaning
`QCHAT_ERR_GEN_*`	-1 … -10	A null, an invalid value, a range, a buffer too small, not found, a wrong state, a timeout, full, exists, denied
`QCHAT_ERR_MEM_*`	-101	Memory could not be reserved
`QCHAT_ERR_CFG_*`	-201 … -210	The configuration: file, path, port, address, limit, census, console, log, network, store
`QCHAT_ERR_IDN_*`	-301 … -311	The identity: missing, unusable, unsolved, unsaved, exists, cancelled, export, import, passphrase, format, keyring
`QCHAT_ERR_STO_*`	-401 … -405	The local store: open, write, read, format, full
`QCHAT_ERR_NET_*`	-501 … -504	The node: refused, send failed, not ready, no such peer
`QCHAT_ERR_LGR_*`	-601 … -611	The ledger: create, conversation, entry, history, key, key pending, forked, member, role, invite, signers
`QCHAT_ERR_ROOM_*`	-701 … -709	A conversation: kind, name, not a member, full, closed, exists, malformed id, not found, read only
`QCHAT_ERR_MSG_*`	-801 … -808	A message: format, decrypt, no key, not a member, too big, hidden, not a chat entry, not found

Family	Range	Meaning
`QCHAT_ERR_KEY_*`	-901 ... -905	The reading keys: wrap, unwrap, no peer key, format, a primitive failed
`QCHAT_ERR_PRF_*`	-1001 ... -1005	A profile: format, signature, size, stale, unknown
`QCHAT_ERR_PRS_*`	-1101 ... -1103	Presence: format, kind, the switch is off
`QCHAT_ERR_BLK_*`	-1201 ... -1206	The explorer: chain, height, transaction, proof, off, page too large
`QCHAT_ERR_FIL_*`	-1301 ... -1309	Files: drop, vault, path, name, size, pending, rotated, not found, incomplete
`QCHAT_ERR_WS_*`	-1401 ... -1409	The surface: token, unauthenticated, schema, action, argument, connections, remote, send, JSON
`QCHAT_ERR_CTL_*`	-1501 ... -1504	The client: connect, reply, denied, usage
`QCHAT_ERR_UPD_*`	-1601 ... -1609	The updater: fetch, manifest, signature, hash, version, apply, format, unsigned build, already current
`QCHAT_ERR_WEB_*`	-1701 ... -1702	The console host: document root, start
`QCHAT_ERR_SUB_*`	-1801 ... -1809	A component underneath refused
`QCHAT_ERR_SYS_*`	-1901 ... -1906	The system: mutex, clock, file, signal, thread, socket

Known limits

- A profile is learned from a contact and from nobody else; there is no directory to look a person up in.
- The vault keeps its fragments on this machine only.
- Only the founder rotates the key of a conversation, and a late joiner is handed the current epoch and no earlier one.
- The explorer is read on the machine the chain lives on.
- This build carries the placeholder update key until the release key is pinned into it.
- The web console is documented in its own repository once released.