Omni-IRC UI Client contract (mini-spec)

Principles

- **One-way dataflow:**
  - UI sends **commands** to client (/msg, /join, /list)
  - Client **sends IRC Command** to Server (`PRIVMSG :~ , JOIN #channel, etc…`)
  - Client **detects IRC Events** from server.
  - Client mutates **authoritative state**
  - → Client emits `CLIENT {json}` **snapshots**.

    All **Send/Event Handler** actions are Async with Lwt

- **Separation of concerns:** Raw IRC fanout (e.g., `PRIVMSG, NOTICE, etc…`) **is not part of CLIENT blobs**; UIs must handle `ClientRxChunk` directly.for IRC i/o
  - `CLIENT {json}` **snapshots**.Only convey **STATE of client**.

```
CLIENT
{"type":"client_user","op":"pointer","nick":"jesse","key":"jesse"}

CLIENT {"type":"user","op":"upsert","user":{ /* sanitized fields */ }}
```

- **Stateless messages:** Every `CLIENT {json}` is a *momentary snapshot* (idempotent, mergeable on the UI side).

Commands (UI -> Client)

- `UiCmd(key,args)` where `key` maps via `Cmd_key.t`.
  - Examples: `JOIN`, `NAMES`, `WHOIS`, `SELF`, `GET_LIST`, `CHANNEL`, `RAW`, …
- UI never mutates state directly; it only “pulls levers”.

The Client maintains its own state to simplify the task for implementations..Applications that use omni-irc can **use the client** for mundane IRC related calls, **get the state from the core,** and focus only on code that would be relevant to the application; not mundane IRC housekeeping.

Client events (Client -> UI)

- `CLIENT { ... }` lines (JSON) for **state snapshots**:
    - `{"type":"channels","op":"snapshot"|"upsert"|"remove", ...}`
    - `{"type":"channel", "channel":{...}}`
    - `{"type":"chanlist", "entries":[...], "filter":..., "limit":...}`
    - `{"type":"user","op":"upsert","user":{...}}`
    - `{"type":"client_user","op":"pointer","nick":"<nick>","key":"<norm>"}`
        - Optionally followed by a `user` **upsert** for that nick: `{"type":"user","op":"upsert","user":{...}}`

Required UI behaviors

- Maintain its **own** state mirror; treat every `CLIENT` blob as *authoritative* delta/snapshot.
- For `client_user`:
    - Update the “self” pointer (`self_user_key := key`, etc.).
    - If a `user` upsert arrives, merge it into `users`.
- For channel ops:
    - Apply `snapshot`/`upsert`/`remove` exactly (idempotent).

Required Client behaviors

- Mutate internal models on IRC/Core events.

- When meaningful state changes occur, **emit a `CLIENT` snapshot** appropriate for the change.
- Provide **commands** that trigger snapshots on demand (e.g., `/self`, `/channel #x`, `/list`).

Invariants & guardrails

- **Idempotency:** repeating the same `CLIENT` blob must not corrupt UI state.
- **Schema stability:** include `"type"` and small, stable field names; add `"op"` when actions differ.
- **No back-pressure surprises:** throttle very hot emissions (e.g., huge `NAMES`) by batching or truncating with explicit "(… N more)" markers (you already do this).
- **UTF-8 & control-codes:** sanitize exactly once (client already does; UI should be tolerant).

Example flow: "Who am I?"

1. UI: user types `/self` → `UiCmd("SELF", [])`.

Client: handles `SELF` → emits:

```
CLIENT
{"type":"client_user","op":"pointer","nick":"jesse","key":"jesse"}
CLIENT {"type":"user","op":"upsert","user":{ /* sanitized fields */ }}
```

2.
3. UI: sets `self_user_key := "jesse"`, merges `user` upsert.

Minimal test plan

- Connect with explicit `--nick`, verify client emits `client_user` on connect (or after `/self`).
- Change nick: client updates internal `self` pointer and emits new `client_user` + `user` upsert.
- Join/part triggers channel upserts; UI reflects counts and topic without touching raw PRIVMSG paths.
- WHOIS cache refresh: `user` upsert emitted without extra network request if fresh.